



CROCODILE[®] User Manual

The Cisco Router Configuration Diligent Evaluator

Authors:

Reinhard Schwarz
Holger Peine
Thomas Schwenkler
Kai Simon

IESE Report-Nr. 011.04/E
Version 3.4
October 2008

Eine Publikation des Fraunhofer IESE

Fraunhofer IESE is an institute of the
Fraunhofer Gesellschaft.

The institute transfers innovative software
development techniques, methods, and tools
into industrial practice, assists companies in
building software competencies customized
to their needs, and helps them to establish a
competitive market position.

The Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Fraunhofer-Platz 1
67663 Kaiserslautern
Germany

Preface

In the following, we will describe the evaluation tool CROCODILE[®]. The tool analyzes router configurations and identifies potential security weaknesses implied therein. CROCODILE thus supports the user in performing security revisions of computer networks.

Goals and audience of this manual

This manual primarily addresses the user. It gives an overview of handling the tool and of its evaluation and support functions. Furthermore, CROCODILE users are sufficiently familiarized with the tool's structure and its basic program operation to be able to master minor application problems on their own.

The manual also addresses not-yet users of the tool — that is, people who are interested, but not yet sure if CROCODILE might be a useful building block for their security solution. We would like to provide the undetermined reader with sufficient information about the tool's capabilities and limits to make an informed decision either in favor of or against the CROCODILE approach.

Potential developers intending to modify or extend the tool on their own should read the user manual as an introductory document before tackling any system work. In favor of pure users, we have purposely refrained from delving too deep into implementation details. In particular, we present only a few program fragments in this document in order not to overstrain readers with internals of the Perl programming language. If necessary, the user may safely skip the concerned parts of the text without losing the plot. Nevertheless, the developer will find a lot of useful information about the tool's implementation in this manual.

General prerequisite knowledge

It was our intention to conceive a tool that is easy to install, intuitive to handle, and compatible with different execution platforms as far as possible. Basic computer knowledge should suffice to successfully start using the tool. As the typical CROCODILE user works with router configurations, it seems safe to assume basic abilities in dealing with computers and software.

The tool is started from the command line, usually from a user shell, a console window or similar. Accordingly, users should be familiar with the command line interface of their operating system. On our part, we are not limited to a specific platform: The tool was developed partially under dialects of Unix (Solaris, Linux), and partially under various variants of Microsoft Windows.

The evaluation results are formatted for display by an HTML browser. As HTML documents are displayed differently by different browsers, it may be required now and then to fine-tune some browser settings. For instance, users should know how to change font sizes and activate »HTML Frames« or »Cascading Style Sheets« in their browser. We deem this reasonable in the age of the Internet. Further display parameters, in particular color settings, may be configured within CROCODILE. The manual describes the steps required for this.

Necessary router knowledge

CROCODILE analyzes router configurations, or more exactly configuration descriptions for routers running the Internetworking Operating System (IOS) by Cisco Systems. It goes without saying, then, that the user must be familiar with routing problems and IOS in order to profit from the tool. Implementing adequate measures in reaction to the evaluation results requires routing expertise in any case.

The tool employs evaluation criteria that are partially fixed and partially configurable. In order to formulate their own evaluation criteria, users must possess sufficient router and security knowledge. As security requirements vary widely across different deployment contexts, CROCODILE cannot provide a useful universal default configuration; the included sample configurations serve mainly for illustration and will presumably produce some meaningless evaluation results. The configurable evaluation criteria thus require an adaptation to the local context. The user manual will explain the necessary steps for customization in the concerned places.

Modification and extension of the tool

CROCODILE was conceived as an extensible framework, meaning that the functionality of the tool is not fixed, but may easily be extended or modified. The user may assume the developer's role whenever required by conceiving additional — or completely different — checker modules and plugging them into the framework¹. The user manual provides basic insights for developers — a

¹ If desired, we offer to perform such extensions or adaptations on a contract basis.

starting point to explore the structure and functionality of the tool. However, it cannot replace a genuine developer manual or a developer training by Fraunhofer IESE.

When performing tool extensions, programming experience in the Perl programming language is indispensable. This manual provides an outline of the tool's architecture and the construction scheme of checker modules. Assuming sufficient programming experience, the existing checker modules may serve as prototypes for custom modules, if the user has purchased a source code licence for CROCODILE. (The tool is also offered in an »obfuscated version« with disguised source code.)

New features in CROCODILE Version 3.3

Compared to earlier versions, CROCODILE Version 3.3 provides the following improvements:

- **More expressive evaluation rule description language**

A considerably extended syntax to formulate custom evaluation rules is available now. In addition to logical aggregation of simple evaluation criteria to compound rules with AND, OR, XOR, and IF ... THEN ... ELSE, the rule syntax now also supports variable binding with universal or existential quantors, and also user-defined evaluation messages.

- **Improved links to online information**

The link database with pointers into Cisco's *Online Reference Manuals* has been extended considerably, so that more and better hyperlinks can be provided for arbitrary configuration commands.

- **More compact reporting, improved readability and navigation**

The report generator has been improved to better suppress redundant annotations. For better guidance, hyperlinks to separate CROCODILE views are more systematically colored depending on the severity of the linked content. The bookmark structure has been simplified. In the interactive version, the hypertext evaluation report is now represented by fewer HTML files, although it provides more content and more hyperlinks.

- **RAT 2.2 compliance**

CROCODILE's emulation module for the well-known *Router Auditing Tool* (RAT), originally conceived for RAT 2.0, has been upgraded to support the latest RAT Benchmark Version 2.2 [RAT2.2], released in September 2004.

- **HTML 4.01 compliance**

To achieve maximum portability, all HTML and CSS output that **CROCODILE**

generates has been validated against the HTML 4.01 and CSS2 specifications with validation services provided by the WorldWide Web Consortium (W3C), the Web Design Group (WDG), and others. All pages were found to be in valid »HTML 4.01 Transitional« or »HTML 4.01 Frameset«² format. Thus, any modern HTML browser should be capable of reproducing a proper hypertext evaluation report.

- **Additional command line options**

We added command line options for the selection of specific evaluation modules, and for suppressing XML output where it is not required.

New features in CROCODILE Version 3.4

The latest CROCODILE Version 3.4 offers the following improvements:

- **Improved result database**

The internal interfaces for storage and retrieval of evaluation results was cleaned up and improved. All classes related to result data objects were put into the subdirectory 'Database'. The new database classes improve the readability of the code and provide better support for future framework extensions.

- **Bookmark and context directives for compound patterns**

To improve the structure of evaluation reports, evaluation rules specified as so-called »compound patterns« (Section 4.2) may now assign their findings to user-defined logical contexts. Furthermore, related contexts may be grouped under a common, user-defined bookmark for more convenient browsing of the interactive report. See Section »Assigning evaluation results to contexts and bookmarks« for more details.

- **Support for evaluation message catalogues**

Optionally, a catalogue of evaluation messages may be provided, and evaluation messages may be tagged so that CROCODILE can automatically generate hyperlinks from evaluation reports to more detailed background information in the message catalogue. The catalogue defines a common reference document for quality assurance. Refer to »Optional: Adding background information to evaluation results« in Section 8.3 for more details.

² There is one single exception where a »Frameset« page slightly deviates from strict HTML 4.01 compliance: Starting page 'index.html' uses the frameset attributes 'border', 'frameborder', and 'framespacing' to suppress borders around the bookmark and the main window frames. These violations of the standard were necessary because most current browsers (e.g., Internet Explorer or Mozilla) do not properly support official HTML 4.01 syntax. This is a well-known problem, so we followed the recommended work-around that is supported by most browsers.

More information

This manual provides detailed information about CROCODILE and its application. However, it is only a static, textual description, enhanced with some screenshots that illustrate the key concepts of the tool. To better convey the tool's »Look & Feel«, we refer the reader to the CROCODILE home page at

`http://www.iese.fhg.de/crocodile`

This page offers an »Online Demonstration« where the user can explore CROCODILE's graphical user interface. In addition to that, it provides the latest news about the tool and an up-to-date version of this manual for downloading.

Contact

This manual cannot answer all questions surfacing in practical use of CROCODILE. We do hope, however, to provide a sufficient basis, enabling the users to discover the subtle details of the tool on their own.

In order to further improve usability and utility of CROCODILE, we are glad to hear any suggestions from our audience. Comments about the tool or its documentation may be directed to the following address:

`crocodile@iese.fraunhofer.de`

A corresponding link — 'Contact' — is incorporated in the hypertext output of the tool.

Inhaltsverzeichnis

Preface	v
1 Problem and Proposed Solution	1
1.1 Network security by secure router configurations	1
1.2 Tool support for router security revisions	1
1.3 Security evaluation criteria	2
1.4 Structure of this manual	3
2 Getting Started — CROCODILE in a Nutshell	5
2.1 System requirements	5
2.2 Installing and removing the CROCODILE tool suite	5
2.3 Choosing the input data	6
2.4 Running CROCODILE	7
2.5 Displaying the evaluation results	9
2.6 Run-time options and help function	10
2.7 The next steps ...	11
3 Framework Architecture and Internal Operation	13
3.1 Overview	13
3.2 Parser module	17
3.3 Checker modules	25
3.4 Result database	29
3.5 Evaluation reports	32

3.6	Auxiliary modules	38
3.7	Directory structure	38
4	Checker Modules	41
4.1	Selecting checker modules for execution	41
4.2	Module 'CompoundPatterns'	42
4.3	Module 'IngressEgress'	58
4.4	Module 'Routing'	65
4.5	Module 'Connectivity'	66
4.6	Module 'SNMP'	72
4.7	Module 'NTP'	74
4.8	Module 'AAA'	77
4.9	Module 'Logging'	80
4.10	Module 'NAT'	82
4.11	Module 'Passwords'	83
4.12	Module 'CiscoLinks'	86
5	Emulation of the Router Auditing Tool (RAT)	89
5.1	Emulation module 'RATemulation'	89
5.2	RAT conformity	91
5.3	Configuring the emulation module	92
5.4	Benchmark computation (CIS benchmark and CROCODILE score)	94
5.5	Evaluation reports	95
5.6	Rule descriptions	97
5.7	Concluding remarks	97
6	Batch Processing	101
6.1	Using CROCODILE in batch processing mode	101

6.2	Configurable options	105
6.3	Choosing evaluation rule sets	105
6.4	Machine-readable result formats	109
6.5	Logging the batch processing	109
7	Utilities	111
7.1	Analyzing filtering properties of ACLs and interfaces using 'blackwhite'	111
7.2	Reformatting the evaluation report using 'make_html'	113
7.3	Providing URLs to the IOS Command Reference	116
8	Creating Custom Checker Modules	121
8.1	Information gathering	121
8.2	Analysis	122
8.3	Implementing the evaluation logic	124
8.4	Polishing	134
8.5	Remarks	135
9	Status and Outlook	137
9.1	Status of the tool	137
9.2	Next steps	137
9.3	Extension to other network components	138
Appendix A — Interface to the Result Database		139
References		143
List of Abbreviations		145

1 Problem and Proposed Solution

IP-based communication networks have gained in importance over the last few years. More and more enterprises are connected to the Internet, exposing their information technology (IT) infrastructure to »attacks from cyber space«.

1.1 Network security by secure router configurations

The security of IP networks is substantially influenced by the routers relaying IP packets between senders and receivers. Modern routers perform not only relaying functions, but serve also to filter, separate, and audit data streams. On top of that, the router's operating system usually contains numerous standard services — such as telnet, tftp, or snmp — that may easily cause hazards for the network components.

Routers must be configured with great diligence in order to counter threats by eavesdropping, jamming, forging or redirecting IP traffic and to ensure that attackers cannot use the router as a bridgehead for further attacks on the network. However, configuring a router is a difficult task.

1.2 Tool support for router security revisions

Router configurations may be very complex and hard to understand in all their ramifications. Accordingly, there is a high risk of a network administrator committing configuration errors that slip manual inspection. Adequate tool support can significantly contribute to discover such hidden weaknesses in router configurations.

CROCODILE is such a checking tool. It reads router configurations in text format and scans them for potential shortcomings. The analysis is based on evaluation criteria that are partly fixed and partly configurable. The results of the analysis are reported as a hypertext document.

The evaluation tool is currently geared at routers by Cisco Systems running the IOS operating system. This class of routers dominates today's market. However, even though all evaluation criteria refer to IOS router configurations, the evaluation tool has been conceived as a modular, IOS-independent framework. It is perfectly possible to support other types of devices — such as Cisco PIX firewalls

— in the same fashion. Applying the CROCODILE framework requires nothing more than a configuration description in text format, as a sequence of individual configuration clauses.³

The following requirements have governed the design of the tool support:

- **Automation**
The tool should be simple to use and it should minimize the effort required for tool configuration and application.
- **Configurability**
The tool should be simple to adapt to changing security requirements and evaluation topics.
- **Portability**
The tool should run on various platforms, in particular on common Unix and Windows systems. Installation and deinstallation should be painless.
- **Modularity**
The tool should be simple to extend — including extensions by the user.

The following chapters will explain how these requirements are mirrored in the design of the evaluation tool.

1.3 Security evaluation criteria

The benefit delivered by CROCODILE is primarily determined by the quality of its evaluation criteria. The criteria employed by the tool are derived from publicly available recommendations for secure router configuration.

The most important sources for such security recommendations known to us are provided by the vendor Cisco. These include various volumes of the Cisco IOS Reference Library, in particular [IOS12.0a] and [IOS12.0b]. Furthermore, Cisco offers various publications for download, for example [CiscoTech]. The central configuration baseline for public network operators is the Cisco recommendation »Essential IOS Features Every ISP Should Consider« [IOSessentials]. The »Essentials« are frequently updated and are arguably the best representation of what can be considered good practice in this area.

³ In contrast to this, network components that are configurable solely by a graphical user interface — without an option to export their configuration settings in textual format — would pose a problem. Some router vendors store configuration attributes solely in proprietary binary formats. The CROCODILE approach would be unsuitable in this case.

The current »Router Security Configuration Guide« of the U.S. National Security Agency (NSA) [NSAGuide] offers very detailed information as well. It contains detailed configuration examples recommended for IOS routers. [NSAGuide] and [IOSessentials] cover largely the same ground, with the NSA guide frequently describing the necessary actions somewhat more illustratively and precisely. However, the two sources disagree regarding certain configuration recommendations. This is because the requirements of public network operators can hardly be reconciled with those of the operators of private, local area networks. Security requirements must always be considered in the context of the concrete application!

We will abstain from detailing the employed evaluation criteria in the following. In Chapter 4, we sketch the available checker modules and their respective evaluation scope, without further elaborating the meaning of individual checks. The reader is referred to the sources named above for such information, and to the hyperlinks inserted by CROCODILE into the evaluation report.

1.4 Structure of this manual

This manual describes the properties, the architecture, and the operation of the CROCODILE tool. Chapter 2 gives an overview of how the tool is used and what system requirements it assumes. Chapter 3 describes the structure of the software and the underlying design strategy, sketching the interplay of the various tool components.

Chapter 4 is concerned with the available checker modules, describing their operation and scope. Chapter 5 is dedicated to a specific checker module that emulates the Router Auditing Tool (RAT), a well-known configuration checker freely available for download in the Internet.

Chapter 6 describes the batch processing capabilities provided by CROCODILE's 'batchcroco' front end. Chapter 7 presents various auxiliary tools complementing the tool.

Chapter 8 addresses developers planning to extend the tool by further checker modules. A rough overview about the procedure to create new checker modules is given.

Finally, Chapter 9 offers a short summary and an outlook on future developments.

2 Getting Started — CROCODILE in a Nutshell

Before introducing the user to the manifold features of the tool by a lengthy description, we would like to briefly summarize the essential steps in using the tool. This quick overview should enable the user to install the tool and perform a simple standard evaluation run.

2.1 System requirements

Hardware and operating system

The tool does not require any special or sophisticated hardware. A custom Intel-based PC with a clock rate of at least 100 MHz, 128 MByte of main memory, and a few MByte of disk space running under Linux or Windows is sufficient.

The tool is by no means limited to Intel platforms, but runs without problems on other architectures as well. Actually, CROCODILE was developed and tested primarily under Sun Solaris on a SPARC processor.

Perl run-time environment

As the tool is implemented in the Perl programming language, it requires a Perl run-time environment for execution. Perl is included in any standard Unix or Linux distribution. On Windows platforms, prepackaged Perl executables — or optionally source code archives — are freely available on the Internet.

The CROCODILE software was developed using the Perl versions 5.4, 5.6, and 5.8, and tested successfully on Sun Solaris 2.6 and 8, RedHat Linux kernel 2.4, and Windows 2000 operating systems. To ensure the portability of the code, we have avoided the use of Perl extension modules exceeding the scope of a standard distribution.

2.2 Installing and removing the CROCODILE tool suite

We assume that Perl has already been installed. The tool itself does not require any special preparations apart from that. For instance, on a Unix platform the user must perform the following steps in order to install CROCODILE:

1. Login and move to a free directory, for example, the '/tmp' directory

```
user@host [~]      cd /tmp
user@host [/tmp]
```

2. Copy the tool archive from the installation media (e.g., a floppy disk) and unpack it

```
user@host [/tmp]   cp /floppy/crocodile.tar .
user@host [/tmp]   tar xvf crocodile.tar
```

3. Move to the tool subdirectory

```
user@host [/tmp]   cd CROCODILE
```

The tool is now ready for use. To remove the installation, it is sufficient to remove the 'CROCODILE' subdirectory (or whatever installation path was chosen).

CROCODILE does not access any data outside its installation directory, unless the user explicitly specifies different input or output file paths.

2.3 Choosing the input data

CROCODILE expects an IOS router configuration in common text format as input, as produced, for example, by the IOS command 'show running-config'. The »running config« is ideal for this purpose, as

- it will always show the true configuration state of the router during operation;
- all IOS keywords are spelled out in full length (which is not necessary with manual configuration)

A sort of shortcoming of the »running config« is that default settings are not normally shown explicitly.

Nothing prevents feeding a manually written configuration to CROCODILE, as the tool assumes neither the specific output order of 'show running-config' nor its characteristic mode-dependent indenting of the configuration text. If an externally stored configuration description is used as a data source (such as a file obtained from a TFTP server), it must be ensured that all keywords are spelled out completely, as CROCODILE ignores— with a few exceptions — any abbreviations. Additionally, the user must ensure that the configuration under evaluation does indeed match the one in operation.

Accordingly, we recommend to obtain the »running config« of the fully configured router in any case and subject that to a final evaluation run.⁴

2.4 Running CROCODILE

The tool directory contains the main program 'crocodile', as well as several subdirectories as shown in Figure 10, page 39.

In the 'SampleConfigs' subdirectory, some example configurations are available for demonstration purposes. A configuration description may be analyzed by feeding it into the Perl program as a simple text file as follows⁵:

```
user@host [CROCODILE] perl crocodile SampleConfigs/acsac.cfg
```

This command feeds the file 'acsac.cfg' as input to the Perl program 'crocodile'⁶. Status and error messages are written to the console.

The example assumes that the Perl interpreter can be found in the user's search path. Otherwise, the complete path name of the Perl installation must be given instead of only 'perl', such as '/usr/local/bin/perl'. After starting the program, output similar to the following will appear on the console:

```
CROCODILE started.

Evaluation Target:  SampleConfigs/acsac.cfg

Reading '/tmp/CROCODILE/Configuration/plugin.conf' ...
... done (plugin.conf)

Reading '/tmp/CROCODILE/Configuration/linkdata.conf' ...
... done (linkdata.conf)

Reading '/tmp/CROCODILE/Configuration/designators.conf' ...
... done (designators.conf)

Reading '/tmp/CROCODILE/Configuration/IngressEgress.conf' ...
... done (IngressEgress.conf)

Reading '/tmp/CROCODILE/Configuration/Passwords.conf' ...
```

⁴ Apart from that, it should be checked which configuration the router will use after rebooting: Is the »boot configuration« really the current »running config«?

⁵ The syntax 'perl crocodile <input file>' is the canonical form that should work on any platform. Depending on the local operating systems, the user may shorten the command to 'crocodile <input file>', provided that the program file 'crocodile' is linked with the Perl program path of the local system. See the header line of the main program 'crocodile' for an example!

⁶ The program is designed so as to optionally read from standard input, too. Under Unix, the following commands are possible: (1) Piping within a process chain — 'cat <input file> | crocodile', (2) Input redirection — 'crocodile < <input file>'

```

... done (Passwords.conf)

Reading '/tmp/CROCODILE/Configuration/CompoundPatterns.conf' ...
... done (CompoundPatterns.conf)

Passwords PLUGGED IN.

CompoundPatterns PLUGGED IN.

Connectivity PLUGGED IN.

NTP PLUGGED IN.

IngressEgress PLUGGED IN.

AAA PLUGGED IN.

SNMP PLUGGED IN.

Logging PLUGGED IN.

Starting parse ...
    114 lines of configuration parsed.

Parsing finished. Now analyzing ...

Checking for trivial No. 5 passwords ...

Dumping subnet information to result database ...

Dumping node information to result database ...

Writing interface info to '/tmp/CROCODILE/XML_data' ...

Writing line interface info to '/tmp/CROCODILE/XML_data' ...

Writing access-list info to '/tmp/CROCODILE/XML_data' ...
    Skipping 'ACL 110' (empty!): Used but not defined?

Determine whether interfaces accept  NTP requests:
    Serial0
    Ethernet0
    -- undefined ACL '110' --

Ingress Restrictions for Interfaces ...

Egress Restrictions for Interfaces ...

Ingress Restrictions for Lines ...

Egress Restrictions for Lines ...

Restrictions for Accesslists ...

```

```

Reading '/tmp/CROCODILE/Output/HTML_output.conf' ...
... done (HTML_output.conf)

Writing Results ...

Analysis finished. See '/tmp/CROCODILE/HTML_results/index.html'.

Successful termination: No PROBLEMS and no ERRORS.
user time 0:19 min    system time 0:00 min

```

The analysis of 'acsac.cfg' requires only a few seconds, depending on the performance of the available hardware. To indicate the progress of the computation, a line number or completion percentage quote increasing with the progress of the evaluation may appear among the console output.

2.5 Displaying the evaluation results

The results of an evaluation run are provided in different views as hypertext in HTML format in the directory 'HTML_results', starting from the root page 'index.html'. A standard HTML browser suffices for display, such as Netscape Navigator, Mozilla, Internet Explorer or Konqueror (Figure 1).

To structure the hypertext documents, the tool uses HTML frames as well as a central Cascading Style Sheet available as 'HTML_results/default.css'. Relying on these formatting aids achieves a clear and flexibly adaptable document structure. However, the employed browser must be suitably configured in order to use the style sheet settings instead of the browser default settings. Modern browsers should smoothly support frames as well as style sheets. The exact settings to achieve this, however, depend on the specific browser.

Note: *An unpleasant, but still mostly usable display format will result with incorrect browser settings. To achieve well balanced window sizes, easily recognized color settings, and amenable font sizes, the user must choose suitable browser settings to have the provided style sheet evaluated correctly. Microsoft Internet Explorer 6.0, for instance, offers the required settings under 'Tools fi Internet Options ...' and then in the 'General fi Accessibility' tab.*

Users may customize the key parameters of the HTML format — such as font sizes and color settings — to their preferences by changing the defaults in the file 'Output/HTML_output.conf'. An adjustment may become necessary, as different browsers (and color displays) interpret color codes or font sizes sometimes rather differently. After changing the display parameters corresponding to the local system platform, an existing evaluation result can be redisplayed with the help of the 'make_html.pl' utility (see Chapter 7.2).

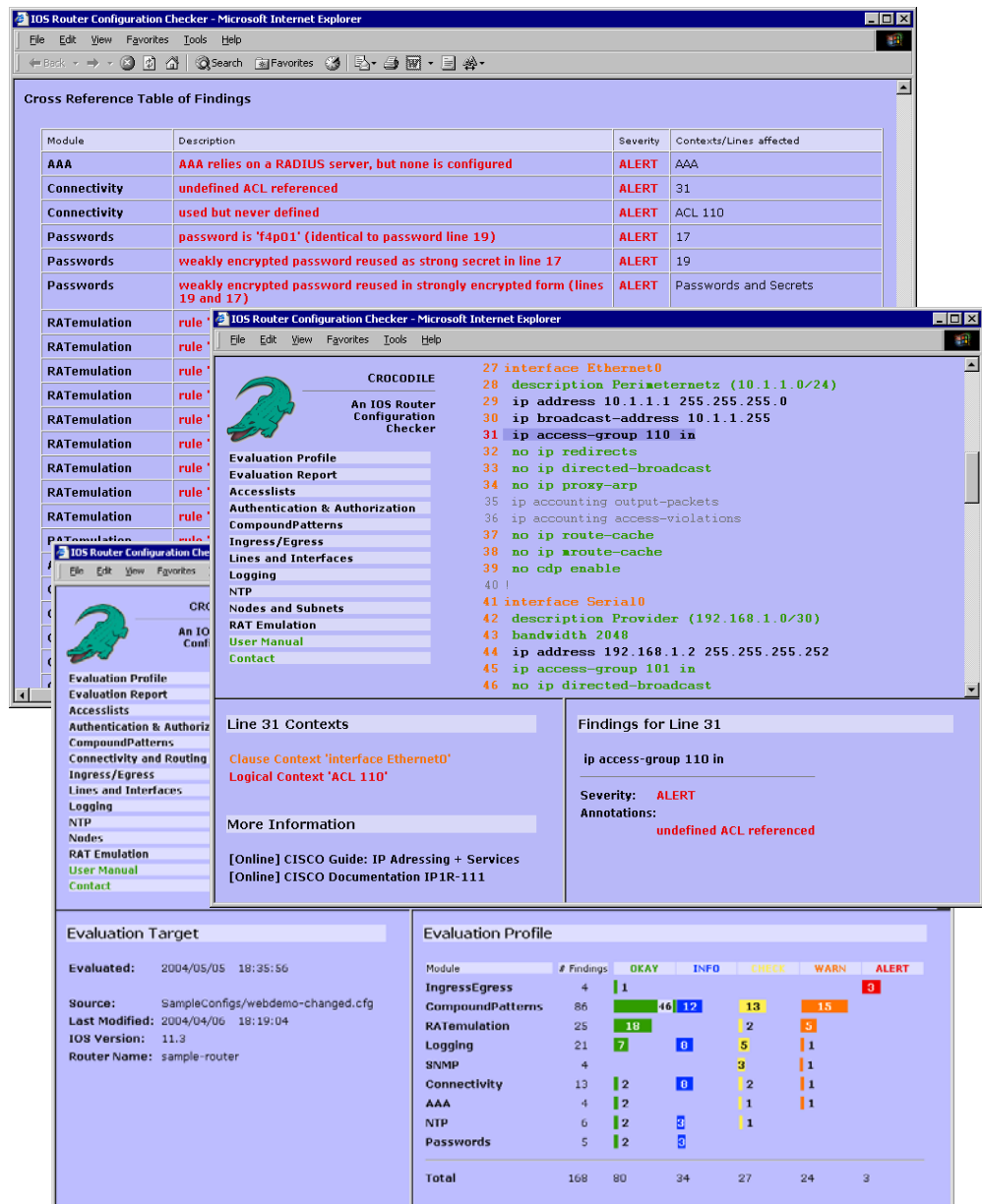


Figure 1 Evaluation results as HTML hypertext

2.6 Run-time options and help function

The execution of the 'crocodile' main program may be influenced by various invocation parameters. For example, the level of detail of the console messages (»verbosity«) may be increased in steps for debugging purposes, and all status messages may be logged to a file (»logging«). A description of all supported run-

time options would lead too far at this place. For an initial invocation, no special command line parameters will normally be required.

CROCODILE is mostly self-explanatory. It provides an online help function (option '-h') for quick orientation. A short description of the run-time options may be obtained, for example, using the following invocation:

```
user@host [CROCODILE] perl crocodile -h
USAGE

crocodile [ <options> ] [ <router config file> ]

DESCRIPTION

CROCODILE reads a router configuration file in ASCII format,
analyzes the configuration settings, and creates evaluation reports
in ASCII, HTML, or XML format, depending on command line options.
If no file name (or filename '-') is specified, STDIN is read.
OPTIONS

The options below are general option for both interactive and
batch mode.

-c <dir> Configuration directory where CROCODILE *.conf files can be
        found; default is './Configuration'

-h          Show this help information on console and exit

... further options follow ...
```

Part of the options are marked as »for internal use only«, to run 'crocodile' in batch processing mode (see Chapter 2.7). The user may safely ignore these run-time parameters in interactive operation.

2.7 The next steps ...

After becoming basically familiar with 'crocodile', the user may consciously select some checker modules in order to analyze router configurations using these modules. Custom checking rules may also be formulated. Details of how to configure CROCODILE's checking capabilities are described in Chapter 4.

Apart from the interactive analysis of individual configuration files, CROCODILE also supports the evaluation of a series of configurations in batch mode. A separate tool front end is used for batch processing: 'batchcroco'. This program also provides an online help function with a short description of the available invocation parameters (option '-h'):

```
user@host[CROCODILE] perl batchcroco -h
```

USAGE

```
batchcroco [ <options> ] <router config files or directories> ...
```

DESCRIPTION

This is the batch processing front end for CROCODILE. It reads in a set of router configuration files in text format, and feeds them into CROCODILE for analysis. The results of all evaluations are collected. See './crocodile -h' for more details.

OPTIONS

```
-d <dir>          Store output in <dir> (default: current directory)
```

```
-f <fmt>          Generate output format <fmt>: ASCII or HTML
```

... further options follow ...

A detailed description of the batch processing front end is given in Chapter 6. All command line options may also be pre-configured in the configuration file 'Configuration/batch.conf'; this configuration file contains detailed explanations of the format and meaning of the individual configuration parameters.

3 Framework Architecture and Internal Operation

This chapter describes the structure and workings of the CROCODILE tool. The design is oriented towards maximum flexibility in order to adapt the tool to new or changed evaluation criteria with minimum effort.

3.1 Overview

Tool components

The tool can be roughly structured into three types of components: One pattern matching module (parser), an arbitrary number of checker modules, and a display component. Each *checker module* covers a self-contained evaluation topic with all implied logical checking and interpretation steps required to arrive at an evaluation result. The *parser* constitutes the link between the checker modules and the input data. The *display component* generates suitable graphical presentations of the evaluation's findings and prepares a hypertext evaluation report in HTML format.

Overall operation

The evaluation tool works according to the following schema. The parser reads a valid Cisco configuration file in text format. It matches the input text line by line with a set of predefined text patterns. Whenever one of the patterns matches a line from the configuration text, the parser notifies all checker modules that have registered for the concerned pattern, and hands them the text line matching the pattern for evaluation.

When a checker module receives a notification, it executes a pattern handler routine associated with the pattern. The pattern handler is a method of the checker module and comprises all logical checking steps to be performed on occurrence of this pattern. The method invocation returns a checking result and optional comments about that result. All results are fed in structured form to an internal result data base.

After all input lines have been processed in this way, each checker module is asked one last time by a special method invocation to return a conclusive overall

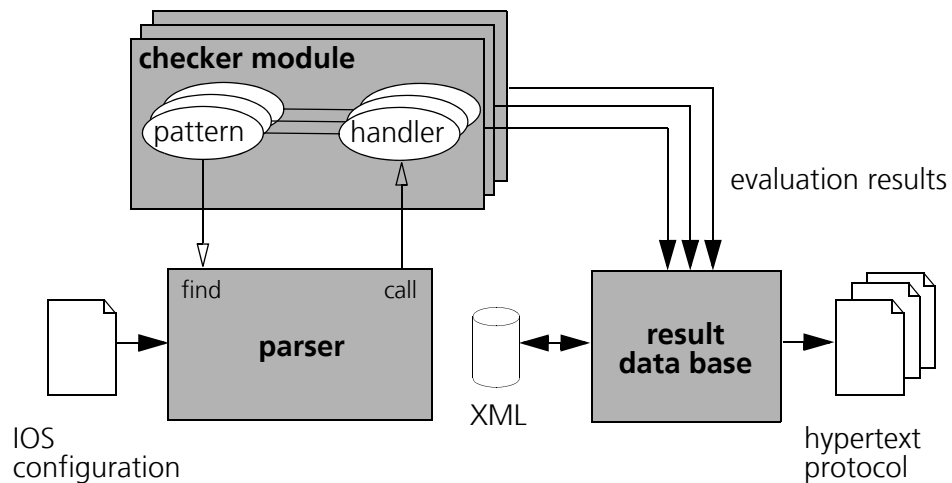


Figure 2 Overview of structure and operation of the evaluation tool

judgement of its evaluation topic. This enables the checker modules to build an integrated view of their respective topic that takes into account multiply pattern occurrences and is not restricted to isolated input lines.

The final results returned from all checker modules are added to the findings collected in the result data base. After completion of the parser run, the whole result data base is stored as a file in XML format. This raw data is then used to generate various HTML views that may be displayed with an ordinary web browser. Figure 2 shows the principle of operation in overview.

Checker modules as «software plug-ins»

The evaluation tool is prepared for execution by »plugging« the desired checker modules into the parser. All checker modules possess a standardized interface matching exactly with a corresponding plug-in interface in the parser. Checker modules are basically independent from each other. They may be plugged into the parser module individually or in groups to configure an evaluation run with an individual scope.

The user is free to either provide a separate checker module for each security aspect to be checked, or to gather logically related issues in a common module. In the extreme case, there is a separate checker module for every single text pattern. Usually, however, a checker module performs the analysis of several, sometimes dozens of text patterns all pertaining to an overall logical security topic. Further, nothing prevents the same text pattern from being analyzed by several checker modules, each time under a different aspect. The choice of module granularity should be determined solely by the needs of the user. The

extent of a module is mostly determined by the clarity and comfort of programming as well as the ease of tool application.

Of course, individual checker modules may also cooperate, for example by building and maintaining common intermediate result data. However, it must be noted that in this case a single module may no longer be used independently of the other modules it cooperates with. The evaluation result is complete and correct only if all cooperating modules have also participated in the evaluation run and contributed their respective findings. For this reason, it is advisable to minimize module interdependence when designing new checker modules. Every checker module may declare a list (`»REQUIRED_MODULES«`) of all checker modules it depends on. The evaluation tool will enforce the plug-in of all `REQUIRED_MODULES` even if the user forgets to plug them into the parser explicitly.

Program implementation

The evaluation tool was implemented in the Perl (Practical Extraction and Report Language) programming language. Several reasons motivated this choice:

- Perl is freely available and runs on the most varied hardware platforms and operating systems. In particular, Perl run-time environments are readily available for the popular Linux and Windows systems, allowing painless porting of the tool.
- Perl provides very powerful and very efficient operations for text manipulation, pattern matching, and the like. Program code for text parsing and text analysis can thus be formulated elegantly and compactly without harming the execution speed.
- Perl is object-oriented. Components of a specific type can thus inherit common traits from a based type. For instance, all checker modules inherit their plug-in interface to the parser from a common base class: `'Plugin::Module'`; when new checker modules are created, no dedicated program code needs to be written for interfacing with the CROCODILE framework, significantly reducing the programming effort.
- Perl programs are usually executed by an interpreter, requiring no prior compilation or linking. This facilitates rapid prototyping when creating new checker modules.

The tool's components — parser, checker modules, and display module — are integrated into a common application by the 'crocodile' main program (Figure 3). Basically, the main program performs the following steps:

```

print "\nCROCODILE started ...\n\n";

# Name and path of the various input and output files
my $input = "-";
my $html_outdir="./HTML_results";
my $xml_outdir = "./XML_data";
my $xml_output = "$xml_outdir/results.xml";

# Create and initialize result display
my $html=HTML_control->new("Output/HTML_output.conf");

# Create and initialize the parser: plug in checker modules
my $parser = Parser->new($input);

# Create and initialize one instance of each configured checker module
my @modulelist = plugin_config();
my ($modulename, $module);

foreach $modulename (@modulelist) {
    require "Plugin::$name";
    $module = $modulename->new();
    $parser->plugin($module);
}

# Parse input file line by line
print "\nStarting parse ...\n\n";
$parser->parse();

# Create conclusive evaluation
print "\nParsing finished. Now analyzing ...\n\n";
$parser->analyze();

# Write result data in various formats
print "\nWriting Results... \n";
my $data=ResultDB->get();
$data->save_as_xml($xml_output);
$html->write($html_outdir);

print STDERR "\nAnalysis finished.\n\n";

```

Figure 3

Structure of the CROCODILE main program

1. First, the file names for input and output must be determined. (In contrast to Figure 3, invocation parameters would normally be used for this. The above example omits this for the sake of clarity.) The various HTML and XML output formats are generated by convenient Perl objects with suitable support functionality.
2. One instance each of the parser and all desired checker modules are then created. The checker modules are then plugged into the parser by the

'plugin()' method.

3. Now the parser is ready to start. The text is parsed line by line, and the resulting findings are written to the result database by the concerned checker modules.
4. After parsing is finished, the parser performs a conclusive analysis with all connected modules, with each module drawing an overall conclusion from all its individual findings for its specific topic of analysis.
5. The program finishes with writing the analysis data in various text, XML, and HTML formats.

As can be seen, the program structure supports the extension of the tool by new checker modules in a very straightforward way, simply by including their name in a configurable list of plug-ins. In particular, there is no need to modify existing CROCODILE code. And as Perl is executed by an interpreter, not even re-compilation or re-linking of the software is required. Deriving newly created checker modules from the Perl class 'Plugin::Module' ensures that they can easily be integrated using 'plugin()'.

3.2 Parser module

A basic function required by all plug-in modules is finding certain text patterns in the router configuration description, and decomposing them into their syntactical building blocks. The parser module performs this task on behalf of all plug-ins. When a module is plugged into the parser using 'plugin()', it passes a set of pattern specifications to the parser that the parser is to detect on behalf of the module.

If one of the indicated patterns occurs in the configuration text, the parser notifies the checker module and passes it the text line matching the pattern for further processing, decomposed into its constituents according to the pattern specification.

Patterns

Plug-in modules specify their relevant text patterns in Backus-Naur format. A pattern description is composed of one or more fundamental building blocks (atoms) that may be combined to compounds. The constituents may be combined according to the following rules:

- **Sequence**

The sequence '*<pattern1> <pattern2> ...*' makes a new pattern.

- **Alternative**
 '{ <pattern1> | <pattern2> | ... }' means:
 »either <pattern1> or <pattern2> or«
- **Option**
 '[<pattern>]' means: »The <pattern> in brackets is optional.«
- **Repetition**
 '{ <pattern> }*' means:
 »The <pattern> in braces repeats one or more times.«

 '[<pattern>]*' means:
 »The <pattern> in brackets is missing or repeats arbitrarily often.«
- **Negation**
 '!<pattern>' means: »The <pattern> does not occur (in this place).«

Available atoms

A number of predefined atoms is provided for specifying patterns. Further atoms may easily be added to the parser if need be. The currently available atoms and their meaning are listed in Table 1. For each atom, an equivalent Perl regular expression is given where possible, in order to formally define which patterns are recognized by the parser as a valid match.

A typical pattern specification might be

```
access-list {1-99|1300-1999}{permit|deny}{any|IPADDR [IPMASK]} [log]
```

In this pattern, 'access-list', 'permit', 'deny', 'any' and 'log' are reserved words of the Cisco configuration language, that is, »IOS keyword« atoms. 'IPADDR' denotes the atom for an IP address. 'IPMASK' is a macro, predefined as a synonym for 'IPADDR'. In general, atoms (and macros — see below) can be recognized by upper case letters, while reserved words are spelled in lower case with an optional initial capital. The range specifications '1-99' and '1300-1399' denote an arbitrary number from that range. The above pattern, for instance, matches the following texts:

```
»access-list 3 permit any«  
»access-list 1300 deny 143.69.0.0 0.0.255.255 log«
```

The pattern description was consciously chosen such that its format corresponds to that of relevant Cisco manuals. The user should have few problems creating the pattern corresponding to the syntax of a given IOS configuration clause.

Atom identifier	Regular expression	Description
STRING	<code>^\s.*\$</code>	Rest of line, starting with first significant character
WORD	<code>^[^\s!]+</code>	Character string without white space (without any appended comment)
NAME	<code>^[A-Za-z][\w.-]*</code>	Word made of letters, dots, dashes, and underscores
NUM	<code>^\d+</code>	Integer number
IPADDR	<code>^\d+\.\d+\.\d+\.\d+</code>	IP address in dot notation, with each number from the range 0–255
DSTRINGD	<code>^D[^D]*D</code>	Character string enclosed in delimiter character D (may span multiple lines)
...	<code>^[^!]*</code>	Match the remainder of the line (without any appended comment)
PATTERN	–	Match a valid BNF Pattern according to CROCODILE syntax conventions
<code><range></code> NUM- NUM	<code>^\d+</code>	Match an integer number from the specified interval <code><from> – <to></code>
<code><numeral></code> NUM['.' NUM]*	–	Match the specified integer or floating point number
<code><literal></code> <code>' ([^']*) '</code>	–	Match an identical character string enclosed in single quotes
<code><IOS keyword></code> <code>^[A-Za-z-][a-z-]*\$</code>	–	Match an identical IOS keyword in arbitrary case
<code><makro></code> <code>^[A-Z_] [A-Z_\d]*\$</code>	–	Will be substituted by the corresponding CROCODILE pattern (see Section »Macros«, page 23)
<code><regular expression></code> <code>/(([^\^\/\ \.] \\.) *) /</code>	–	Match an identical Perl regular expression (with an implicit <code>'^\s*'</code> prefix: "Start match at first significant character")

Table 1 Predefined atoms to form CROCODILE patterns

The »regular expression« type of atom enables the use of native Perl expressions within a pattern. Perl regular expressions support very powerful constructs with little effort. For example, the pattern

```
interface /( ?i) [a-z]+/ /\d+ (: \d+ | \ / \d+ ( \ . \d+ ) ? ) * / [ ... ]
```

will match any of the following text lines:

```
»interface eth 0«
»interface ATM3/0.1 point-to-point«
»interface Serial2/1/7:20«
```

However, Perl expressions should not be used without good reason if equivalent CROCODILE atoms exist: Regular expressions are hard to read, and errors easily slip into such specifications.

Pattern decomposition

Whenever an indicated pattern matches a line of input, the parser notifies all concerned checker modules and passes them the corresponding piece of text. To facilitate the analysis by the checker modules, the parser provides the data for each such text excerpt in a convenient form. The information passed includes

- the **line number** where the text was found in the input stream
- the **text line** in its original form
- the **pattern** that triggered this notification of the checker module
- the **text fragment** corresponding to each pattern constituent

The constituents (atoms or compounds) are hierarchically numbered in order to map them unambiguously to the corresponding text fragments. The example from above would result in the following numbering scheme:

```
access-list {1-99|1300-1999}{permit|deny}{any|IPADDR [IPMASK] }
0:      access-list
1:      {1-99|1300-1999}
1.0:    1-99
1.1:    1300-1999
2:      {permit|deny}
2.0:    permit
2.1:    deny
3:      {any|IPADDR [IPMASK] }
3.0:    any
3.1:    IPADDR
3.2:    [IPMASK]
```

The schema of this numbering should be obvious: The constituents of the pattern are numbered increasingly from left to right, starting at zero. Every opening brace or bracket starts the next lower level of numbering, where the same scheme is

applied recursively. A closing brace or bracket results in a return to the next upper numbering level.

When a text line matching this pattern is found, the parser assigns each text fragment to the number of its corresponding pattern fragment. For instance, the 'access-list' clause from the example above would result in the following decomposition:

```
»access-list 1370 deny 143.69.0.0 0.0.255.255«
0:      'access-list'
1:      '1370'
1.1:    '1370'
2:      'deny'
2.1:    'deny'
3:      '143.69.0.0 0.0.255.255'
3.1:    '143.69.0.0'
3.2:    '0.0.255.255'
```

The analysis methods (handlers) of the checker module may simply refer to the fragment number of the pattern in order to access specific text fragments. The numbering allows a regular access to arbitrary pattern specifications.

Conflicts: Greedy or minimal pattern interpretation?

Depending on the pattern, there may be more than one way to match a given text. This is because the pattern constituents may be applied »greedily« or »minimally«.

The »greedy« matching strategy tries to assign a text string as long as possible to an atom before attempting to apply the subsequent atom to the remainder of the string. The minimal strategy, on the other hand, tries to assign only as short a string as possible to each atom. Arbitrary blends of these two extremes are conceivable.

An example will illustrate the problem. Suppose we have specified the pattern

```
enable secret [level NUM] STRING
```

and apply it to the following text:

```
»enable secret level 15 5 $1$iNeB$tr.JFRdI7ckzIX8v9g8c1«.
```

Two interpretations of the patterns are conceivable here:

- The optional pattern constituent '[level NUM]' is applicable and may thus match the substring »level 15«. The atom 'STRING' will then be assigned the remaining text, that is »5 \$1\$iNeB\$tr.JFRdI7ckzIX8v9g8c1«.
- However, another way to match would be to skip the optional pattern constituent '[level NUM]'. In this case, STRING will match »level 15 5 \$1\$iNeB\$tr.JFRdI7ckzIX8v9g8c1«, with the optional '[level NUM]' matching the empty string »«.

None of the strategies — neither greedy nor the minimal pattern matching — is preferable to the other in principle; both have their uses. However, for CROCODILE it is important that a pattern always has a unique interpretation. To ensure this, an interpretation strategy for all patterns could be determined at will. However, this would easily result in usage errors, as users will often not notice ambiguities in their patterns, and they may intend an interpretation that is neither greedy nor minimal. To avoid misunderstandings, CROCODILE solves the problem in another way: The CROCODILE parser checks all possible interpretations of a pattern; if two distinct interpretations for a pattern are possible, CROCODILE will report the ambiguity. In the above example, the following message would be reported:

```
*** Ambiguous pattern caused two conflicting pattern interpretations.
*** Pattern: 'enable secret [level NUM] STRING'
*** Text:    'enable secret level 15 5 $1$iNeB$tr.JFRdI7ckzIX8v9g8c1'
*** Match1:
***          0                'enable'
***          1                'secret'
***          3                'level 15 5 $1$iNeB$tr.JFRdI7ckzIX8v9g8c1'
*** Match2:
***          0                'enable'
***          1                'secret'
***          2                'level 15'
***          2.0              'level'
***          2.1              '15'
***          3                '5 $1$iNeB$tr.JFRdI7ckzIX8v9g8c1'
*** Conflicting attribute '3':
***   interpretation 1 is 'level 15 5 $1$iNeB$tr.JFRdI7ckzIX8v9g8c1'
***   interpretation 2 is '5 $1$iNeB$tr.JFRdI7ckzIX8v9g8c1'
```

This message means that component no. 3 of the pattern — that is, 'STRING' according to the canonical numbering schema — may be assigned two different matches when applied to the given text

```
»enable secret level 15 5 $1$iNeB$tr.JFRdI7ckzIX8v9g8c1«.
```

The two alternative interpretations are shown.

To make the evaluation run finish without error, the user must refine the objectionable pattern such that a unique interpretation results. In the given case, the following refinement might be useful:

```
enable secret [level NUM] !level STRING
```

In this version, the optional part '[level NUM]' may be omitted only if the remainder of the text line does not start with the keyword 'level'. This solves the ambiguity problem. A foolproof solution, though probably unnecessarily complex in this example, would be

```
enable secret [level NUM] !{level NUM} STRING
```

This version has the advantage of being derivable in a schematic way from the objectionable pattern without much thinking.⁷

Macros

Some IOS configuration clauses are very complex and varying. Consequently, the corresponding pattern becomes rather lengthy. In order to specify even complex text patterns as clearly as possible, macros may be used. A macro is a pair (<macro name>, <macro pattern>) with the macro name consisting of capital letters, digits, and underscore characters, with the initial character not being a digit (see Table 1). After its definition, a macro may be used like an atom. Whenever a pattern specification '... <macro name> ...' is used, the pattern {<macro pattern>} is silently substituted in this place. Note that every non-atomic macro pattern is surrounded with braces before the substitution. This ensures that the substitution is really processed as an atomic unit and cannot cause confusion within compounds. An example for a macro is

```
SUBNET => "{ IPADDR IPMASK | any | host IPADDR }"
```

This macro might be used in a comprehensive pattern in order to simplify its formulation. For example, the macro use

```
deny ip SUBNET SUBNET
```

is equivalent to the explicit pattern

```
deny ip { IPADDR IPMASK | any | host IPADDR } { IPADDR IPMASK | any | host IPADDR } .
```

⁷ Note that 'enable secret [level NUM] ![level NUM] STRING' would not work: The negation of an optional pattern — here: '![level NUM]' — will never match, because the optional part '[...]' will always match at least the empty string.

If macros are used within a pattern, the fragments are numbered (see Section »Pattern decomposition« on page 20) according to the explicit form of the text pattern. This ensures that each pattern constituent — whether within a macro or not — can be accessed easily by the checker modules. In the above example, the first keyword 'host' would receive the number 2.3, and its second occurrence the number 3.3. The text matching the complete 'SUBNET' macro may be accessed under the numbers 2 and 3, exactly as if 'SUBNET' were a predefined atom.

Macros may be nested, that is, other macros defined earlier may be used in the definition of a macro. However, care should be taken not to refer the macro itself in its own definition, or another macro that is defined only later.

Each checker module may define its own macros. Several modules may use the same macro name with different meanings without causing conflicts. All macros are resolved locally, that is, relative to the defining module. Accordingly, local macros are not relevant at the global level of the parser. This ensures that no name collisions can result by adding or extending a checker module.

Apart from local macro definitions of individual checker modules, global macros may also be defined in the parser. This allows the introduction of custom atoms of general usefulness without interfering with the operation of the parser. Globally defined macro names may be locally redefined in the checker modules. Locally defined macros overwrite not only global predefinitions, but even the identifiers of the standard atoms (e.g., 'IPADDR', 'NUM'). This is, however, strongly discouraged. Currently, the following macros are globally predefined (for the user, these macros are next to indistinguishable from atoms):

```
IPMASK ::= IPADDR
IF_TYPE ::= /( ?i) [a-z-] +/
IF_NUM ::= /\d+ ( : \d+ | \ / \d+ ( \ . \d+ ) ? ) */
```

Interface to the main program

The parser module has a very narrow interface to the main program. The only externally visible functions of the parser are 'new()', 'parse()', and 'analyze()'. Their usage is illustrated in Figure 3, page 16. The user need not think about the parser's interface to the main program even when developing and plugging in new checker modules.

Interface to the checker modules

Users do not normally come in touch with the parser and the checker modules, even when constructing their own checker modules. The correct interplay results automatically by inheritance.

The interaction between parser and checker modules is one-way only: All activity originates from the parser. From the point of view of the parser, all checker modules are of the object type 'Plugin::Module'. The fact that the modules are really derived and more refined objects is not visible at the interface between parser and checker modules. Accordingly, the parser only calls the public methods of the 'Plugin::Module' object type. Checker modules inherit this interface from their common base type 'Plugin::Module' without further programming effort. The following functions are provided:

- `name()`
The invocation '`$module->name()`' returns the type name of the checker module, which is mostly used in error messages to indicate the precise module where the problem occurred.
- `requires()`
The invocation '`$module->requires()`' returns a list of all checker modules on which the checker module designated by '`$module`' depends. This list is usually empty. However, when developing new modules, the programmer may indicate logical dependencies between individual checker modules. When a module is activated, the activation of all required modules will automatically be enforced, too.
- `patterns()`
The invocation '`$module->patterns()`' returns all patterns the parser is to find and return on behalf of the module, as well as the name of the pattern handler that performs the further processing on the occurrence of this pattern.
- `macros()`
The invocation '`$module->macros()`' returns all macros the module uses to formulate its patterns. The parser uses this information to expand the patterns to their explicit form.
- `postprocessing()`
The invocation '`$module->postprocessing()`' returns the method the module uses to perform the conclusive analysis of its evaluation topic.

3.3 Checker modules

A topic within a security audit usually comprises several individual probe points. Logically related probe points are normally bundled as a dedicated checker module.

Module structure

All checker modules are derived from a common base type, namely the object class 'Plugin::Module'. By means of inheritance, they are equipped with all essential mechanisms required to fit into the tool framework (see Section »Interface to the checker modules« above). Using inheritance, the user may create new checker modules that fit into the tool framework with minimum effort. Figure 4 shows the basic structure of a checker module. All checker modules are based on this program skeleton.

To enhance the tool's evaluation capabilities, the user creates a new checker module according to the module template (Figure 4). The tailoring towards specific checking tasks requires the following actions:

- In the header line, the user provides a name for the checker module.
- In the »macro definitions« section, local macros may be declared in order to simplify the specifications in the »pattern definitions« section below.
- In the »pattern definitions« section, the user decides which text patterns are to be analyzed and which handlers are responsible for each pattern. One pattern may serve several handlers — and vice versa.
- In the »handler methods« section, a check routine must be implemented for each pattern handler declared. (More on this below in Section »Pattern handlers«).
- In the »postprocessing method« section, a routine may be provided that forms a conclusive overall evaluation based on the sum of all text fragments reported to the module and the information gained from all these. This method may remain undefined.

The changes indicated above are sufficient to create a working checker module. Note that the user does not normally have to change the sections »inheritance« and »constructor«. These passages can be copied unchanged from the code template of Figure 4 for a start. Thus the instructions to create a new checker module basically reduce to two steps:

1. Specify the patterns to be analyzed.
2. Write one handler per pattern to perform the desired analysis of the text fragments found, and — if required — one postprocessing handler to perform a conclusive overall analysis of the complete findings.

```

package Plugin::SomeModule;

# Inheritance: Each checker module „is a“ Plugin::Module
# This makes the interface to the parser available automatically
use Plugin::Module;
use vars qw(@ISA);
@ISA = ("Plugin::Module");

# Private macro definitions of this checker module
my $macroref = [
    SUBNET => "IPADDR IPMASK | any | host IPADDR",
    ...
];

# Pattern definitions this checker module is interested in
# indicating the list of all locally responsible pattern handlers
my $patternref = {
    "line [aux|con|console|tty|vty] NUM [NUM]" =>
        { HANDLER => [ \&_line_handler ], },
    ...
};

# Constructor for instance objects of this checker module
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self;

    $self=$class->SUPER::does_exist();
    if (!defined $self) {
        $self = $class->SUPER::new();
        $self->{MACROS} = $macroref;
        $self->{PATTERNS} = $patternref;
        $self->{POSTPROCEDURE} = \&_final_conclusion;
    }
    return $self;
}

# Handler methods: At least one handler per pattern
sub _line_handler {
    my ($self, $match) = @_;
    ...
}
...

# Post-processing method to create the final conclusion
sub _final_conclusion {
    my ($self) = @_;
    ...
}

```

Figure 4

Generic structure of a checker module

Pattern handlers

Pattern handlers (or »handlers« for short) are methods of the checker module that are called by the parser whenever a corresponding text pattern is found in the router configuration description. The parser supplies two parameters to the handler on each handler invocation:

- The '\$self' parameter refers to the checker module and enables the method to access arbitrary private data of the module during the checks. Thus the method is able to store information across several invocations or to exchange insights with other handlers of the checker module.
- The '\$match' parameter refers to an <attribute, value> table containing all relevant information about the pattern and the corresponding text fragment, including:
 - LINE_NO - the line number where the pattern occurred
 - LINE - the complete text of that line
 - MATCH - the part of the text that matches the pattern
 - PATTERN - the pattern that has triggered this invocation
 - EXPANSION - the explicit form of the pattern, with all macros resolved
 - <fragm. no.> - the constituents of the text fragment, decomposed by fragment numbers (see »Pattern decomposition« on page 20).

With the help of these two parameters, the handler has access to all relevant information required for its checks. Figure 5 shows an excerpt from a typical data record passed from the parser to a pattern handler via '\$match'.

Using '\$match', it is easy to determine which pattern has triggered the invocation (a handler can potentially serve many patterns). The exact content of the router configuration statement and its position within the configuration description are available.

The '\$self' reference provides access to context knowledge during the evaluation of a text fragment. Furthermore, it offers access to all support functions inherited from the base class 'Plugin::Module' for performing the checking steps.

Using the available mechanisms, checking steps can be easily implemented as Perl program code, without requiring closer knowledge about the interplay of

```

0:          'interface'
1:          'Loopback0'
1.0:       'Loopback'
1.1:       '0'
CRITICALITY: 'INFO'
EXPANSION:  'interface
            {/(?i)[a-z]+/ /\d+(:\d+|\/\d+(\.\d+)?)*/}'
LINE:      'interface Loopback0'
LINE_NO:   '22'
MATCH:     'interface Loopback0'
PATTERN:   'interface {IF_TYPE IF_NUM}'

```

Figure 5

Typical example of a '\$match' data record as passed to a handler by the parser

main program, parser, checker modules and input/output. An example of how to construct a new checker module may be found in Chapter 8.

Module configuration

Depending on the evaluation topic of a checker module, certain checking steps may require parameterization. The checking framework thus provides the option to assign one or more configuration files to a checker module.

Content and format of such configuration files are not restricted. Every checker module may be equipped with individually structured configuration information. Due to missing format prescriptions, there are currently no standard access methods for module configuration files. It is the responsibility of the programmer of the respective checker module to provide code for opening and reading the configuration file. The source code of the standard modules 'Passwords', 'IngressEgress', and 'CompoundPatterns' may serve as an example of how to make use of module configuration files at various level of complexity (see Chapter 4).

To aid the clarity of the tool framework, all configuration files should be stored in the configuration directory (see directory structure in Chapter 3.7) dedicated to this purpose, and the file should be named '<checker module>.conf' ('<checker module>.conf<no.>', respectively, when using multiple configuration files for one module).

3.4 Result database

All checker modules store their findings in structured form in an internal database of type 'ResultData'. This database is the basis for the overall conclusions drawn

at the end of the evaluation run. Using a database decouples the checker modules from specific display formats of the output.

Logical model of the result data

Checker modules may store result data of various kind in the result database. Possible types of information are

- Annotations
- Suggestions
- Dumps
- References
- Bookmarks

Annotations are strings containing a comment regarding the security status of a configuration clause. Each annotation is labeled with a severity tag, ranging from 'OKAY', 'INFO', 'CHECK' over 'WARN' to 'ALERT'. This tag indicates whether the annotation refers to a positive, neutral, unclear, or even critical finding. The severity of a finding is highlighted in the output by a corresponding color.

Suggestions offer the option of providing a 'bug fix' with a finding. This will usually consist of IOS configuration commands, and may a comment with further explanation. However, the user is free to leave a plain text as a suggestion.

Dumps are used for more complex analysis results in textual format, line by line with varying levels of indenting. Dumps are needed when a finding exceeds the extent of a short comment. The output may contain hyperlinks to dumps.

References indicate cross-relationships between individual findings and, if needed, external sources of information — such as Cisco's IOS master index. Such references are usually displayed as clickable hyperlinks in the output. The target of a reference may be, for instance, a configuration line, a dump, or an external web page.

Bookmarks do not constitute a finding of their own. They are merely a means of structuring in order to gather references into a logical group. When a bookmark category is opened during display, a choice of references belonging to this category appears. The user may introduce arbitrary bookmark categories and assign individual references to any category, including more than one. See Figure 6 on page 33 for an illustration.

All these types of result data may be created with varying scope:

- Results referring to individual configuration lines
- Results referring to clause contexts
- Results referring to logical contexts

Results referring to configuration lines relate to a single clause in the router configuration description. A checker module may assign an arbitrary number of annotations, references or suggestions to a line. By clicking a configuration line in the hypertext display, the user may view all findings assigned to this line.

Results referring to clause contexts relate to the so-called »Configuration Modes« of the IOS router operating system. Configuration modes form a context enabling certain router settings. Examples include the »Interface Mode« or the »Line Mode«. Many configuration statements are supported only in specific contexts, and a configuration error influences all settings within the same context. Checker modules may gather all lines of such a context into a clause context⁸ and assign common annotations, suggestions and references to this context. The clause context with all pertaining lines and results is displayed as a whole. The severity of a clause context results from the severity labels of all its annotations as well as the severity of the pertaining configuration lines. The configuration mode of an IOS command determines its clause context.

Furthermore, lines may be gathered into arbitrary *logical contexts*, and such contexts may in turn be assigned common annotations, suggestions and references. For example, all declaration clauses of an ACL and all clauses referencing this ACL might be gathered into a global »access-list« context. All configuration errors relating to this ACL may thus be stored and displayed as a unit.

A line from the configuration file may be assigned to any number of logical contexts, but — as determined by its IOS configuration mode — to only one clause context.

⁸ »Line context« would be an even more appropriate name for this, but unfortunately that might cause confusion with the »line interface« configuration mode under IOS, which is why we chose »clause context« instead.

Interface to the checker modules

Checker modules do not use the database interface directly, but via wrapper methods provided by the common base class of all modules — 'Plugin::Module' — with a shortened parameter list. Appendix A describes the available methods.

Storing and loading the result database in XML format

Various result displays may be generated from the result database, independent of the evaluation run. The result database is stored in XML format for this purpose. The XML file may later be read again and be displayed in various formats. CROCODILE provides the display tool 'make_html' for this purpose. The program can be found in the 'Utils' subdirectory (see Figure 10 on page 39). It generates a complex HTML view consisting of several result data frames. Further tools for other output formats — such as Microsoft Excel sheets — are possible, but not currently available.⁹

3.5 Evaluation reports

The result database stores all findings in an internal format. There are various ways to generate an acceptable display from this.

Hypertext output

The evaluation tool currently supports an HTML-based display variant that represents the various views in several interconnected frames. The user may navigate within the result data at different levels of abstraction. Figure 6 shows a typical example. In the top window of the presentation, the configuration file under evaluation is displayed. Individual configuration clauses are colored according to their severity.

Each clause is hyperlinked; a click on a clause will display detailed information concerning that clause in the two bottom windows (in Figure 6, line 17 of the configuration text). In the bottom left window, corresponding references are displayed as hyperlinks then; by clicking on one of these links, the desired information is displayed in the bottom right area. If desired, all clauses of a context as well as the findings for that context may be displayed. Further, corresponding pages of the online Cisco IOS Reference Manual may be directly opened¹⁰ — the respective hyperlinks are automatically inserted into the display by the tool.

⁹ But see the option of producing output in CSV format described in Chapter 6.4, which many other programs can import.

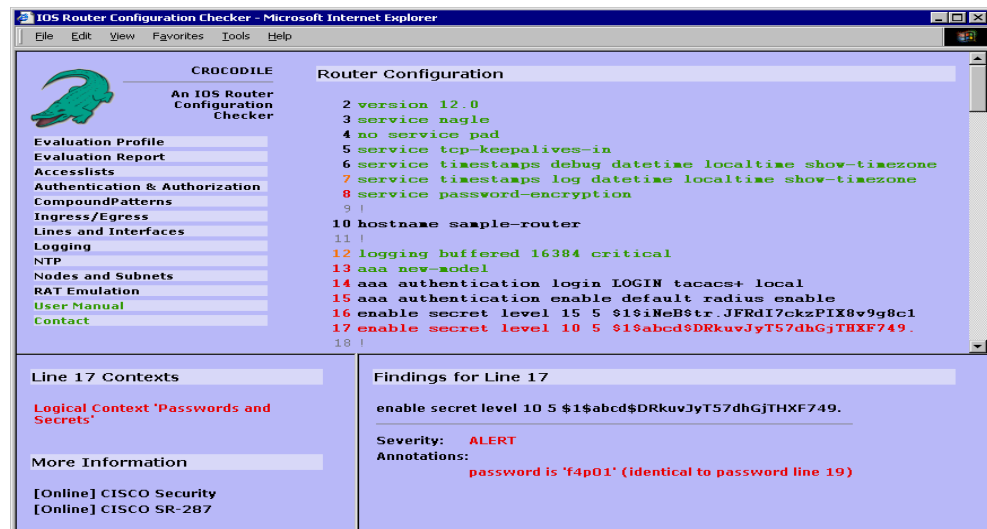


Figure 6 Detailed result presentation in hypertext format

In addition to that, the display provides a 'Bookmark' area. Pointers to more thorough analyses, background information, and to the CROCODILE User Manual are available there. Individual checker modules may create further bookmarks if desired, in order to clearly arrange logically related findings. The tool framework provides a simple programming interface for this purpose (see Appendix A).

HTML overviews

The HTML display is rich in detail, but does require some navigation to collect the sum of all findings. Therefore, the bookmark »Evaluation Report« contains a link to a result overview. The evaluation report is a single HTML page containing the essential annotations about the analyzed router configuration compactly and in context. The user may first skim this display for an initial overview.

Figure 7 shows typical sections of such an evaluation report. The configuration clauses are colored according to their resulting severity. Annotations about the clauses are inserted as comments marked by '***'. In addition to the configuration text the overview contains a cross reference table and an »Evaluation Profile« chart summarizing the evaluation result.

Tip: To determine the origin of an annotation, just move the mouse pointer over the respective line in the »Annotated Configuration Text« or the »Context

¹⁰ To access the IOS Reference Manual, the system must be connected to the Internet.

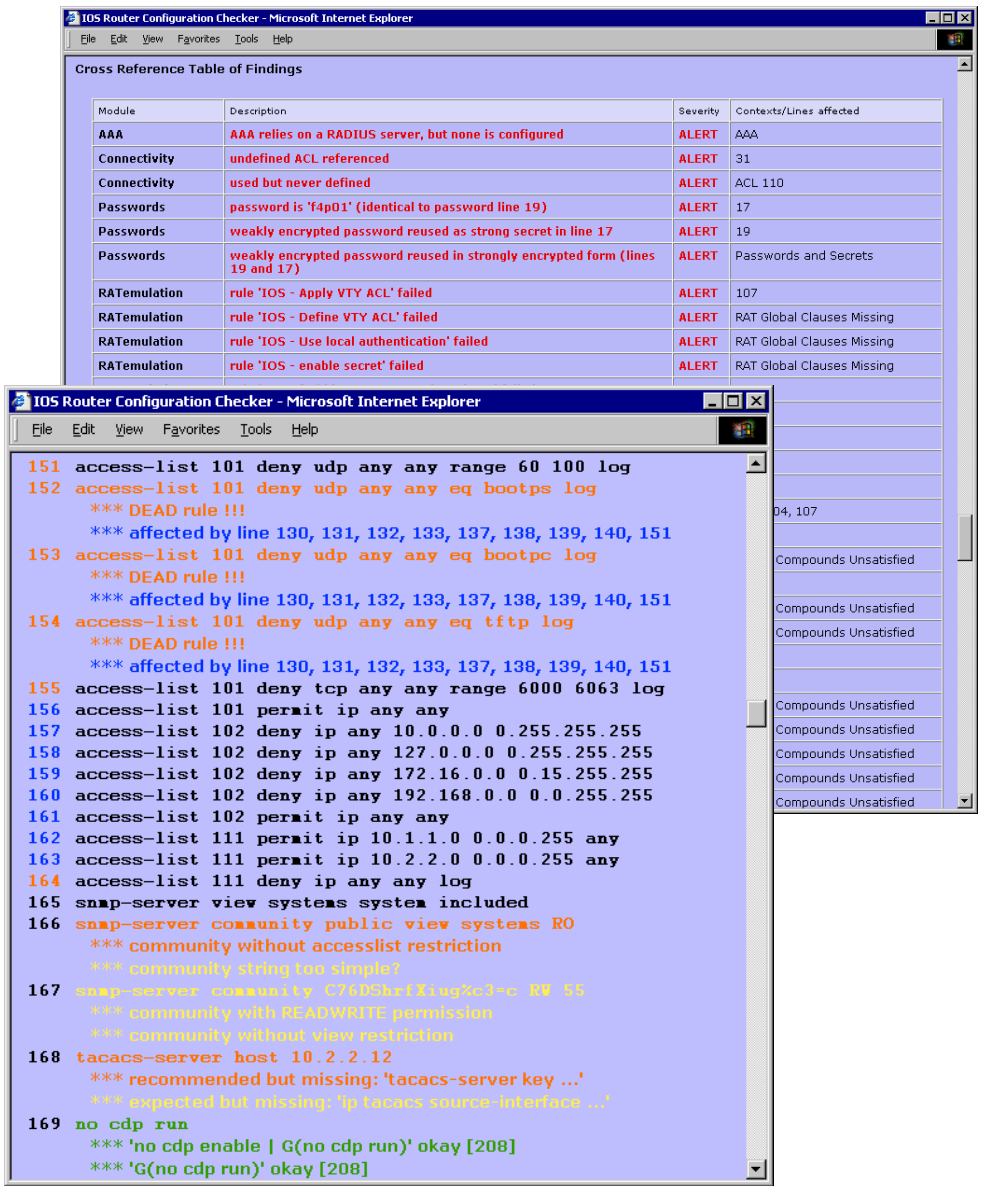


Figure 7 Typical sections of an »Evaluation Report«

Overview« section of the »Evaluation Report«. Provided that JavaScript is enabled, the name of the plugin module that generated the annotation will appear in the status bar of the browser window.

The »Evaluation Profile« view is also directly accessible through the like-named bookmark. Figure 8 shows such an evaluation profile as well as a sample category of findings that can be displayed by clicking into the diagram. This view collects the most important information about the evaluation target and the evaluation

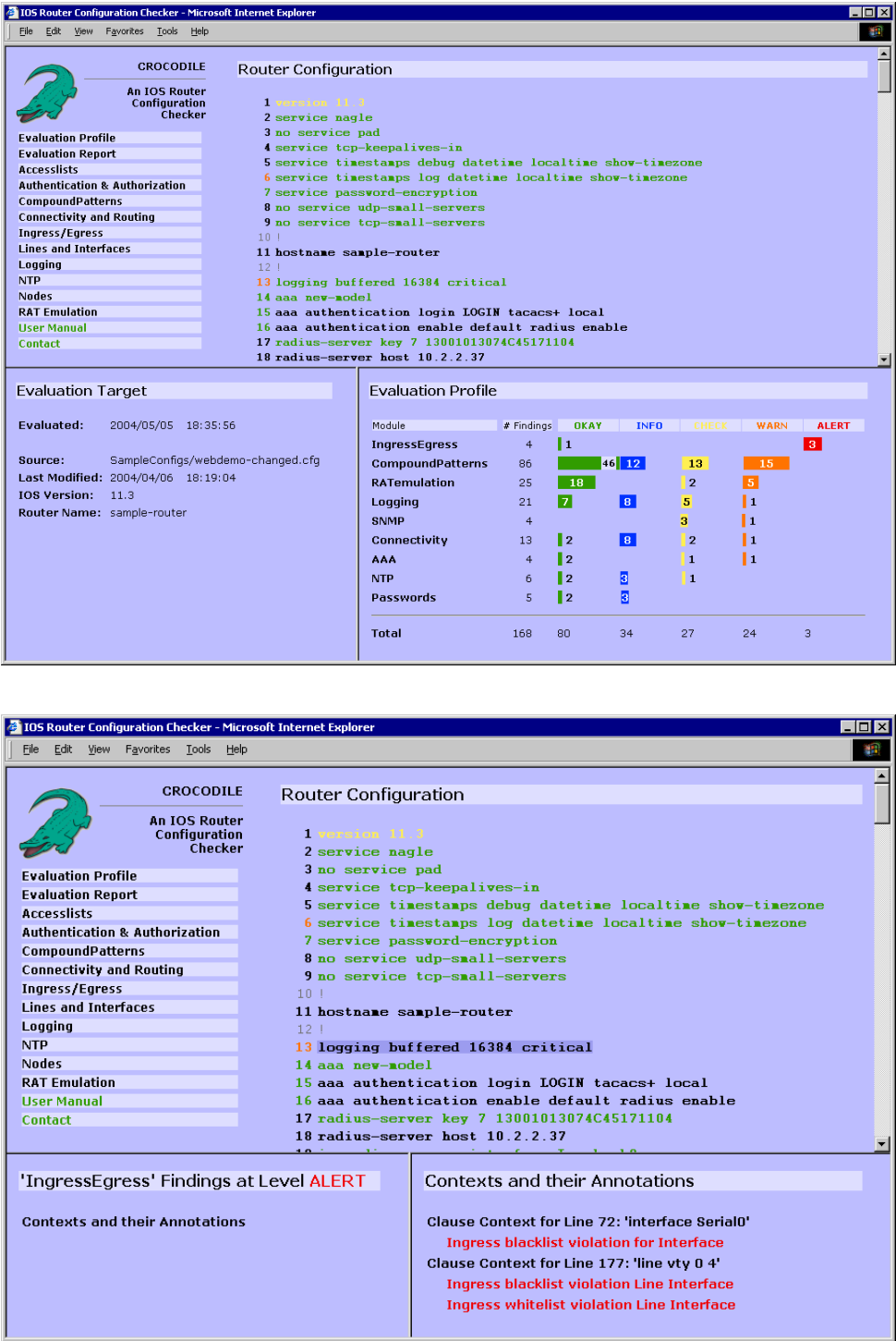


Figure 8 »Evaluation Profile« (top) and detail view of the 'ALERT' result category of the 'IngressEgress' checker module (bottom) as obtained by clicking the corresponding area in the color bar

results. The number and severity of the findings are represented in a bar chart, one bar per checker module. The length of the bars corresponds to the number of findings of the severity categories 'OKAY', 'INFO', 'CHECK', 'WARN' and 'ALERT', respectively, as determined by each checker module. The user thus sees at a glance where the biggest weaknesses of the configuration are hidden. By clicking into the bars, the respective findings may be displayed directly.

Differential result view

Whenever a router configuration is changed, it should be re-examined. In this case the user is less interested in the sum of all findings, but rather in the subset of findings affected by the previous change to the configuration. The tool offers a differential result view for this purpose: In this view, only findings that have changed relative to the previous evaluation result are colored. All other findings that have not changed since the previous evaluation are displayed in a neutral black. Using this mode of display, changes in the evaluation result are immediately obvious. Details about this display mode may be found in Chapter 7.2. One example for a differential result display is shown in Figure 28 on page 117.

Configuration of the display parameters

The HTML generator is configurable. In the file 'Output/HTML_output.conf', the foreground and background colors may be determined, as well as the assignment of colors to the severity levels, preferred font families and font sizes — and some other useful display options. Figure 9 shows the configuration file with some typical settings. The meaning of the individual attributes is usually obvious from their name, with the following exceptions:

The 'useJavaScript' attribute indicates whether CROCODILE generates pure HTML output without embedded script code, or JavaScript-enabled HTML. Enabling JavaScript reduces the amount of output that needs to be generated: The overall data volume is reduced by about 10 percent, and — more importantly — the number of files generated is reduced by about 35 to 50 percent, depending on input and checker rules. For best efficiency, the user should enable JavaScript by default. Apart from a more compact encoding, enabling JavaScript does not affect the content of the generated evaluation report: Information and formatting is the same in both output encoding formats.

The 'show_not_recognized' attribute indicates whether the IOS configuration text is displayed in full, or whether configuration clauses not recognized (i. e., ignored) in the analysis should be suppressed. The 'resize_allowed' attribute determines whether the distribution of the browser window space among the individual frames should be fixed or whether the user is allowed to change the relative frame sizes using the mouse. With the 'frame_borders' attribute the user

```

## HTML_output.conf
## Config-file for HTML output

## JavaScript option (see also command line option '-j')
useJavaScript = YES

## Some general settings about the appearance
show_not_recognized = YES
resize_allowed = YES
frame_borders = YES
main_window_percentage = 55%
link_window_percentage = 35%

## Settings about the style
# You may define a font size, but you may also avoid it
# and just rely on the browsers default settings
# font_size      = 12pt
font_family      = Verdana,Arial,arial,Helvetica,Helvetica
font_size_heading = 130%
font_size_listing = 110%
font_weight      = bold
font_color       = #000
bg_color         = #BBF
heading_bgcolor  = #DDF
margin           = 1px

## Define the RGB-Mix for each severity
colorcode {
    OKAY      = #390
    INFO      = #03F
    CHECK     = #FE5
    WARN      = #F70
    ALERT     = #E00
    UNDEF     = #777
    SOLID     = #000
}

```

Figure 9

Typical settings for the HTML output in 'Output/HTML_output.conf'

may suppress the display of frame borders. The 'main_window_percentage' attribute indicates the share of the total browser window height that is used to display the configuration text; the remainder is used for the frames 'links' and 'display' for the display of evaluation details. The 'link_window_percentage' attribute determines the share of the total browser window width that is reserved for the 'links' frame in the bottom left; the remainder is used for the 'display' frame in the bottom right.

Apart from setting a personally preferred font size, there is normally little need to change the predefined display parameters. Adjustments may become necessary

if outdated display hardware is to be used, or if screenshots of CROCODILE are to be made that are desired in a specific format (e.g., in grayscale).

3.6 Auxiliary modules

When the evaluation criteria become complicated, programming a handler may entail considerable effort. To reduce the complexity of checker modules, the CROCODILE framework encapsulates frequently used functionality in separate auxiliary modules available as utilities to all checker modules. Each newly added checker module may base the implementation of its pattern handlers on a stock of auxiliary functions enabling a compact and clear formulation of the evaluation function. The 'Auxiliary' subdirectory (see Figure 10) contains the auxiliary modules required by the checker modules.

Persistent data

Some checking steps require the evaluation of several router configuration clauses, that is, the combined results of several pattern handlers, possibly even from different modules. Such analyses are only possible if information is stored across the activation of individual pattern handlers. The corresponding data are thus persistent in the sense of existing independently of the duration of individual checker modules until the tool has computed the final evaluation result.

CROCODILE represents persistent data by auxiliary modules. Each auxiliary module is an abstraction of a logical object and provides functions to save and restore relevant object properties, to model relationships between objects, or to apply operators to the object. We recommend using such auxiliary modules as a general means for the structuring of persistent data.

3.7 Directory structure

The source code of the tool is distributed across several subdirectories. The structuring follows the Perl naming conventions: In Perl, module hierarchies are mapped 1:1 to corresponding directory hierarchies. Figure 10 shows the resulting directory structure.

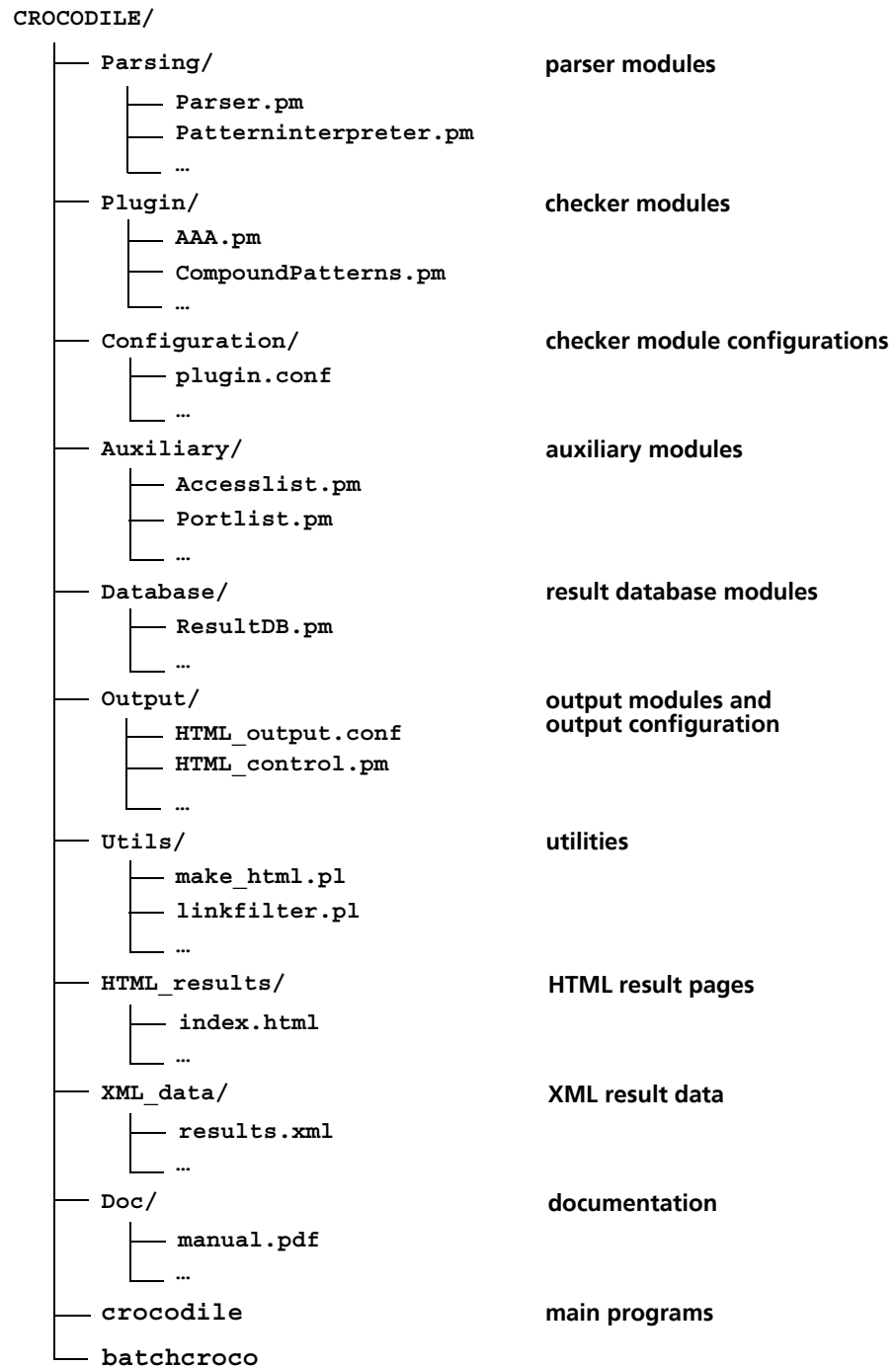


Figure 10 Directory structure of the CROCODILE software

4 Checker Modules

Several checker modules are included with the standard CROCODILE distribution. They implement a number of fundamental security checklist items for IOS configurations. To enhance the tool's capabilities, users may simply plug in their own checker modules (see Chapter 8). Below we describe the modules that are currently provided with the standard distribution.

4.1 Selecting checker modules for execution

The user may choose which of the available checker modules shall be included in an evaluation run. File 'Configuration/plugin.conf' contains the list of modules to be activated. If this file is missing or empty, essentially all modules found under 'Plugin' will be selected.

By deleting (or, preferably, commenting out) individual entries the user may reduce the evaluation scope of the tool. Figure 11 shows an example configuration with the 'NTP.pm' and 'RATemulation.pm' modules deactivated.

```
## plugin.conf -- Which plug-in modules are selected for execution?
##
## If this file exists, only those plug-ins named here are loaded and
## executed. Plug-ins are specified one or multiple per line.
## Leading and trailing whitespace is ignored. A trailing '.pm'
## is optional. Specifications are case-insensitive. Module names
## not found in './Plugin' are ignored.

AAA
CompoundPatterns
Connectivity
IngressEgress
Logging
# NTP
Passwords
SNMP
# RATemulation
```

Figure 11 Configuration example for 'plugin.conf' to select the checker modules to be activated

Individual modules may also be chosen with option '-m <module>' as an command line parameter. This option may be used repeatedly to select more than one module. The name of the module may be abbreviated to a unique prefix, and CROCODILE adds some amount of error correction to guess even misspelt names. If a '-m' option is specified, file 'plugin.conf' is ignored.

4.2 Module 'CompoundPatterns'

The 'CompoundPatterns' checker module is able to cover all evaluation tasks that may be performed by matching simple text patterns in the router configuration description. Security features that depend on certain combinations of single configuration statements without requiring deeper analysis of the clauses' specific parameters may be checked in this way. A considerable number of router configuration features falls in this category.

Configuration of evaluation criteria

'CompoundPatterns' is a universally useful checker module without fixed evaluation criteria. The module reads a list of user-defined evaluation rules from the module configuration file 'CompoundPatterns.conf' (see the directory structure in Figure 10, page 39).

This module illustrates how flexible checker modules may be constructed by parameterizing them with a module configuration file. The exact syntax of evaluation rule specifications is up to the checker module, which may make use of the available mechanisms of the CROCODILE framework, in particular those for pattern matching.

'CompoundPatterns' offers a reasonably simple, yet powerful rule format. The configuration file may contain three kinds of specifications:

- Macro definitions
- Pattern definitions
- Context definitions

Macros

Macro definitions correspond to the usual macros in form and function as they might be defined within a checker module in the 'macros' section (see Figure 4, page 27). They assign symbolic names to complex pattern constituents. The

macro name may later be used as an atom in the construction of more comprehensive pattern definitions. The exact format of a macro definition is

```
<MACRO NAME> ::= <basic pattern>
```

with the macro name formed from a character set comprising capitals, digits, and underscores, with a capital or an underscore as the first character. Two examples might be

```
PROPER_VERSION ::= '12.0' [ '.' NUM ]*
BAD_VERSION    ::= ! PROPER_VERSION NUM [ '.' NUM ]*
```

These definitions mean that every version number starting with '12.0', possibly followed by further subversion numbers separated by '.' is a 'PROPER_VERSION' in the sense of this definition. A 'BAD_VERSION' is every character string that is formed by numbers separated by '.', but does not start with a PROPER_VERSION. This makes, for instance, '12.0.3.34' a PROPER_VERSION, while '12.3' is considered a BAD_VERSION. An application of the two macros is shown in the next section.

Note that only a basic pattern may be used in a macro definition. The use of compound patterns — that is, a combination of elementary patterns using AND, OR, NOT, XOR etc. (see below) — within a macro will be rejected by CROCODILE.

Patterns and Evaluation Messages

Patterns denote text strings of a certain structure in the router configuration description that are to be checked for presence. Form and function of a pattern essentially corresponds to the declarations as they are usually found in the »Patterns« section of a checker module (see Figure 4, page 27). In contrast to pattern declarations in a checker module, however, 'CompoundPatterns.conf' specifies only patterns, not handlers. A handler is not required, as all patterns in 'CompoundPatterns.conf' are processed by a built-in standard handler.

Instead of a handler, the user specifies two different severities per pattern, <sev1> and <sev2>:

- <sev1> indicates how critical the **occurrence** of this pattern in the router configuration description is estimated.
- <sev2> indicates how critical the **absence** of this pattern from the router configuration description is estimated.

The severity estimate consists of a single character. Possible severity values are { o, i, c, w, a } for OKAY, INFO, CHECK, WARN and ALERT as well as '-' for »irrel-

evant«. During the evaluation run, the checker module parses the router configuration for text fragments matching the specified pattern. When such a text fragment is found, the checker module records a severity estimate according to `<sev1>`. If no such pattern occurs in the configuration description, an estimate according to `<sev2>` is recorded. If a '-' is given for `<sev1>` or `<sev2>`, nothing is recorded for the respective case.

By default, the message printed in the evaluation report refers to the text pattern and the line in 'CompoundPatterns.conf' where the pattern is specified; its wording depends on the `<sev1>` and `<sev2>` settings, but cannot be influenced by the user any further unless the user provides an explicit message string for the respective severity. To generate more user-friendly evaluation messages, each severity may be followed by a user-defined message string in double quotes.

The general format of a pattern definition is thus as follows:

```
<sev1> ["<occur message>"] [<sev2> ["<absence message>"]] <pattern>
```

The following examples illustrate some typical cases:

```
aw      ip source-route
a-      ip route '0.0.0.0' '0.0.0.0' Ethernet ...

o ""
c "IOS version declaration expected"
  version PROPER_VERSION

a "Unexpected IOS version specified"    version BAD_VERSION
```

The first pattern definition indicates that the existence of the 'ip source-route' statement in the router configuration must be regarded as highly critical and should result in an ALERT message. The missing of such a statement is considered a shortcoming, too, and results in a WARN message ('no ip source-route' would be most desirable, but the corresponding rule is not included in the example above). As no message is specified, CROCODILE will add a default message text, which is derived from the respective severity and the rule pattern, to the evaluation report.

The second pattern definition indicates that the router statement starting with 'ip route 0.0.0.0 0.0.0.0 Ethernet' is highly undesired and should trigger an ALERT message. If no such clause occurs in the router configuration, this is irrelevant and should not be reported (the hyphen as the second character in 'a-' may be omitted). If the pattern occurs in the configuration, a default alert message is reported.

The pattern definitions three and four have the combined meaning that a proper IOS version number is expected in the configuration description. If the version number is configured as expected, pattern three reports an OKAY message. In this particular case, the user specified an empty message string "", thus the corresponding configuration clause is only marked by coloring it in OKAY color, but no further annotation is added to the evaluation report. If a proper version statement is missing from the configuration description or a bad version number is found, this will trigger a user-defined CHECK or ALERT message, respectively. Note that pattern three has been split into multiple lines to improve readability; CROCODILE tolerates such a pattern layout.

When specifying the severity of a pattern, providing `<sev1>` is sufficient. The second severity will default to '-', as in pattern four of the above example. Basic or compound patterns may be used to define an evaluation rule.

Basic patterns

A basic pattern is a description of a text pattern referring to a single configuration clause — an isolated line of the IOS configuration description. Patterns are specified in Backus-Naur-Form (BNF). The exact format and the possible construction elements (e.g., sequence, alternative, repetition, negation) as well as the available fundamental building blocks (atoms) are described in detail in Chapter 3.2 in Section »Patterns«. Examples of typical pattern specifications may be found there, too.

Caution: *An unfortunately formulated pattern may allow more than one interpretation. Such ambiguities are explained in Chapter 3.2 in Section »Conflicts: Greedy or minimal pattern interpretation?«. If an ambiguous pattern interpretation is recognized, CROCODILE will abort the evaluation run with an error message. The user must then refine that pattern so as to remove the indicated ambiguity.*

Compound patterns

A compound pattern (or shortly: compound) is a combination of several basic patterns linked by logical operators. A nested combination of several (sub-)compounds is also considered a compound pattern. Using compounds, text patterns spanning more than one configuration clause may be specified.

The following operators are available for combination:

- AND-combination `<compound1> & <compound2>`
- OR-combination `<compound1> | <compound2>`

- XOR-combination **XOR**(*<compound1>, ..., <compound n>*)
- Negation **NOT** *<compound>*
- Conditional pattern **IF** (*<condition compound>*)
 [**THEN**] { *<then-compound>* }
 [**ELSIF** (*<cond2 comp.>*) { *<elsif-compound>* }]*
 [**ELSE** { *<else-compound>* }]
- Counting operator **Count**(*<basic pattern>, <minoccur>, <maxoccur>*)

AND, OR, and XOR have their usual meanings: The compound matches if and only if both, at least one, or exactly one of the operands match, respectively. A compound negated by NOT matches if and only if the compound itself does not match.

A conditional compound matches if the corresponding THEN-, ELSIF- or ELSE-compound matches. The choice of the corresponding compound depends on whether the IF-compound or one of the ELSEIF-compounds has matched: The first matching condition compound determines the relevant branch. If none of the condition compounds matches, the ELSE-compound is the corresponding one — if present. If there is no suitable branch, the IF-compound is ignored.

The counting operator Count(...) matches if and only if the operand pattern matches at least *<minoccur>* and at most *<maxoccur>* times. (The upper bound may be omitted, defaulting to »any number of times«). Only a basic pattern may be an operand of a counting operator, since the frequency of arbitrary compound patterns cannot be meaningfully defined in general.

Table 2 gives a formal description of the compound pattern syntax.

Contexts

The IOS configuration language discriminates between different »configuration modes«. Some configuration directives are only available if the router is in a certain IOS mode. To effect such a statement, a prior mode change must be performed. The scope of subsequent statements is limited to the context of the current mode then, as introduced on page 31 as »clause context«.

Clause contexts of a certain type may generally appear in several instances within an IOS configuration description. The interface mode to configure a router interface is a typical example. As a router has several interfaces, all interface configuration statements refer to the interface indicated as the instance at the

<i>Compound</i>	::=	<i>Term</i> [<i>Term</i>]* if (<i>Compound</i>) [then] { <i>Compound</i> } [elsif (<i>Compound</i>) [then] { <i>Compound</i> }]* [else { <i>Compound</i> }]
<i>Term</i>	::=	<i>Factor</i> [& <i>Factor</i>]*
<i>Factor</i>	::=	[NOT] { <i>Pattern</i> <i>Counted_Pattern</i> [Global] (<i>Compound</i>) XOR (<i>Compound</i> { , <i>Compound</i> }*) }
<i>Counted_Pattern</i> ::=		Count (<i>Pattern</i> , <min> [, <max>])
<i>Pattern</i>	::=	{ [!] { <i>Atom</i> <i>Nonatom</i> <macro name> } }*
<i>Nonatom</i>	::=	{ <i>Pattern</i> [<i>Pattern</i>] } [*] [<i>Pattern</i> [<i>Pattern</i>]] [*]
<i>Atom</i>	::=	<IOS Keyword> <numeral> <min>-<max> '<string>' STRING DSTRINGD WORD NAME NUM IPADDR PATTERN ... /<Perl regular expression>/
<i>Macro</i>	::=	<macro name> ::= <i>Pattern</i>

Table 2

Compound pattern syntax. (Characters in boldface are part of the compound pattern, all other characters are metasyms for the syntax description that are not included in the actual compound pattern.)

previous mode change. The same statement may thus repeat several times in the configuration description, once per interface.

To accommodate this concept, it is not sufficient in general to specify the text pattern within a pattern definition. Additionally, the definition must specify in which context and in which context instance the pattern is to match. For example, the occurrence of the 'shutdown' configuration statement can only be meaningfully interpreted if the interface referred to in this statement is clear. Therefore, we have devised the concept of a 'context definition' in the design of the 'CompoundPatterns' module in order to restrict pattern definitions to certain contexts. The context concept described below is even more powerful, as it does not necessarily correspond to an IOS configuration mode; however, in practice the representation of IOS modes is the primary use of this mechanism.

The general format of a context definition is as follows:

```
<begin (basic) pattern> {
    ...
} <end (basic) pattern>
```

Note that *<begin pattern>* and *<end pattern>* must appear in separate lines and not be merged to one line. They must be basic patterns; compounds are neither allowed nor would make any sense here. Between the braces, an arbitrary number of compound definitions may be entered, each starting in a new line. The checker module 'CompoundPatterns' will evaluate these only within the specified context, and ignore them otherwise.

The *<begin pattern>* is an arbitrary basic pattern. It describes the opening clause that IOS uses to mark the beginning of a clause context. As soon as the indicated text pattern occurs in the router configuration description, the context is activated for subsequent lines. Until further notice, the checker module will evaluate the compound definitions given between the braces within the current context instance, processing them like ordinary, context-independent compound definitions. Starting at the clause following *<begin pattern>*, the router configuration is analyzed line by line.

This continues until the *<end pattern>* occurs in the configuration description, another basic pattern. If that pattern matches, the checker module will deactivate the corresponding context as well as all compound definitions local to this context. The context will remain inactive until another text pattern matching the *<begin pattern>* occurs. The *<end pattern>* may be identical to one of the basic patterns inside the braces: In this case, the *<end pattern>* is included in the context defined by *<begin pattern>*.

Activation and deactivation of the same context may thus repeat many times. The compounds within the braces are evaluated separately for each context instance. Multiple contexts may be active concurrently. The following example illustrates the use of context definitions:

```
interface Loopback ... {
    ow if (NOT shutdown) { ip address IPADDR IPMASK }
} interface ...

interface !Loopback ... {
    ow if (NOT shutdown) { no ip redirects &
                          no ip directed-broadcast &
                          no ip proxy-arp
                        }
} interface ...
```

In this example, configuration settings of various interfaces are to be checked, discriminating between loopback interfaces and other interface types.

The first context definition specifies criteria for all loopback interfaces. As soon as a configuration statement starting with the reserved words 'interface Loopback' is recognized, the compound definitions for loopback interfaces are activated. (The three dots '...' appear literally in the compound specification. They form an atomic pattern meaning »arbitrary continuation of this line«). The example checks whether the interface is switched off. If this is the case, the conditional rule does not apply and is therefore ignored. Otherwise, the specification requires that an IP address should be assigned to the interface. If an 'ip address' clause is specified for the interface, an OKAY message is recorded, otherwise a WARN message will appear in the evaluation report.

The loopback definition context ends when a new interface clause is found — no matter what type of interface. If the context end marker happens to be another 'interface Loopback ...' clause, this will deactivate the context only to reactivate it immediately for a new 'Loopback' instance.

The second context definition specifies criteria for all other interfaces apart from loopback interfaces. The start of such a context may be recognized from configuration clauses starting with 'interface' and *not* continuing with 'Loopback'. This context, too, will end when another 'interface ...' clause appears in the configuration description. The clause within the context definition basically amounts to saying that for every interface that is not explicitly switched off, the following features should be defined: 'no ip redirects', 'no ip proxy-arp', and 'no ip directed-broadcast'.

Global context versus clause context

All IOS clauses that do not appear within a clause context are subsumed under the so-called »global context«, and they are called »global clauses«. Global clauses may affect the meaning of context clauses. Therefore, a context-dependent compound often needs to refer to global sub-compounds, too. The 'Global(...)' operator enables references to a global compound from within a context. A typical example might be

```
interface Serial ... {
  ow Global(no cdp run) | no cdp enable
} interface ...
```

This rule expresses that the »Cisco Discovery Protocol« (CDP) for automatic detection of reachable interfaces should be deactivated either at the global level — that is, universally for all interfaces — or separately for all serial interfaces in their respective interface context instance.

Using the Global operator outside of a context specification has no effect on the enclosed compound, but is permissible.

»VIRTUAL« contexts

Remark: *The notion of a virtual context is an advanced feature that is rarely needed in routine applications. The novice user may skip this section on first reading.*

The clause context of an IOS clause reflects the IOS configuration mode to which the configuration command belongs. Each IOS clause occurs in a well-defined, unique configuration mode. Therefore, at most one clause context can be assigned to each clause. CROCODILE enforces the uniqueness of clause contexts.¹¹

Whenever a compound pattern is restricted to a certain context, CROCODILE interprets the context specification as a clause context to which all lines matching the compound specification are added. For example, the compound specification

```
interface Serial ... {
  ow Global(no cdp run) | no cdp enable
} interface ...
```

applied to the IOS configuration

¹¹ Recently, Cisco Systems Inc. have begun to introduce nested configuration modes. Presently, CROCODILE's 'CompoundPatterns' do not (yet) support nested context definitions, but the user will rarely miss this feature.

```
interface Serial 0
  no cdp enable
```

would assign the line 'no cdp enable' to clause context 'interface Serial 0' because the 'cdp' command occurred in the 'interface' configuration mode.

Normally, the user need not care about 'CompoundPatterns' strategy for the assignment of clause contexts: It just works as one would intuitively expect! However, problems may arise if the context mechanism is used to express relations different from configuration modes. For example, after raising the privilege level for 'show ip access-lists' with a

```
privilege exec level 15 show ip access-lists
```

command, the privilege level for the more general 'show ip' should be reset back to level 1 with a

```
privilege exec level 1 show ip
```

clause. The user could specify the following compound to express this fact:

```
privilege exec level [2-15] show ip access-lists {
  ow  privilege exec level 1 show ip
} end
```

The above rule works in principle, but as a side effect CROCODILE creates a clause context 'privilege exec level 15 show ip access-list' and adds 'privilege exec level 1 show ip' as a member! This is quite unsatisfactory because the 'privilege' command belongs to the IOS global configuration mode, so assigning a clause context is rather nonsensical.

To avoid this problem while preserving the flexibility of compound »context« specifications, 'CompoundPatterns' offers the keyword 'VIRTUAL' to mark context specifications as pseudo contexts — that is, as contexts that do not refer to actual IOS configuration modes. Whenever a context restriction is only used to express a sequential order of technically unrelated IOS clauses, a virtual context should be used to prevent CROCODILE from creating a clause context. Thus, the rule above should be refined as follows:

```
VIRTUAL privilege exec level [2-15] show ip access-lists {
  ow  privilege exec level 1 show ip
} end
```

Variable binding with existential and universal quantors

Remark: *This is an advanced topic. Users lacking a mathematical background may only skim this section on first reading. With growing experience, the need for the mechanisms explained below will appear sooner or later, and the user will only then be able to fully appreciate this section.*

With the compound format of Table 2, the user can specify evaluation rules that cover multiple IOS clauses. Unfortunately, the format provides no means to correlate attributes in the individual basic patterns that form the compound.

For example, to express the fact that an access list that is in use must be defined somewhere, the user may try a rule such as:

```
interface Serial ... {
  ow  if (ip access-group NUM ...) { Global(access-list NUM ...) }
}
```

However, this rule will fail because it just states that if *some* access-list is used to protect a serial interface, at least *some* (other?!) access-list must be defined — the rule does not require that both are the same!

What we really mean is something similar to

```
interface Serial ... {
  ow  if (ip access-group X ...) { access-list X ... }
}
```

On second thoughts, this type of refinement has two remaining problems:

- As specified, it is not clear what type of pattern 'X' should match. There is no indication that we expect an ACL number of type 'NUM', or maybe a named access list with an identifier of type 'NAME'.
- The 'ip access-group X ...' command may occur repeatedly, for example once for »in« and once for »out« direction. Is it enough that the rule holds for *some* occurrence of 'ip access-group X ...', or do we require that the rule should be satisfied for *all* occurrences?

To solve the former problem, we have to define the format of 'X'; to solve the latter, we need to add so-called quantors, in mathematical notation:

```
interface Serial ... {
  ow  " x: ip access-group x ... $ access-list x ...
}
```

Module 'CompoundPatterns' provides essentially this feature, but in a more consistent rule syntax. To accommodate quantified variables, we extend the compound pattern format as follows:

- Universal quantor: **foreach** (<macrolist>: <selector>) { <compound> }
- Existential quantor: **exists** (<macrolist>: <selector>) { <compound> }

In these format templates, <macrolist> denotes a comma-separated list of macro definitions according to Table 2, where the macro names serve as variables of the quantor, and the macros define the pattern that the variables must match.

Component <selector> is a basic pattern or a counted pattern (see Table 2) containing some of the macros defined in <macrolist> in unnegated form, potentially with a 'Global(...)' operator applied to <selector>. The selector pattern serves to determine the domain of the values that are bound to the variables by the quantifier: For each IOS clause matching selector, the corresponding values of each macro occurrence in <selector> is determined and »bound« to their macro names, and <compound> is evaluated for each tuple of bound values.

The <compound> in the body of the quantor compounds is an arbitrary type of compound pattern, which is evaluated for each of the selected value tuples. For a quantor to make any sense, <compound> must contain at least one macro name from <macrolist>, or else repeated evaluation yields the same result for every match of <selector>.

Based on this format, we can now rewrite the example above as

```
# For each ACL number (or name) X that occurs in an »ip access-group«
# clause, there is some corresponding ACL rule matching the
# pattern »access-list X ...«

interface Serial ... {
    ow    foreach(X ::= {NUM | NAME}: ip access-group X ...) {
        access-list X ...
    }
}
```

Quantor compounds are first-class members of the compound family, so they may be negated, XOR-ed, and the like. In particular, this means that a quantor compound may occur as a body of another quantor. Thus, quantors may be nested. Moreover, quantors may occur wherever a compound pattern is feasible; the following example is in perfect accordance with compound pattern syntax, although rather nonsensical:

```
# If there exists an arbitrary line X that occurs twice (or more)
```

```
# in the IOS configuration, then raise an 'ALERT' (thereby marking
# every configuration line with 'ALERT' color)

a if (exists (X ::= ...: X) { Count(X, 2) }) {
    ...
}
```

We refer the reader to the IPsec rules in 'CompoundPatterns.conf', for instance, for more reasonable examples of quantor compounds.

In this manual, there is no room to explain the true subtleties of the quantor mechanism. Therefore, we restrict further discussion, and mention in passing just a few useful peculiarities of quantor compounds:

- A typical example for multiple macros within a single quantor is

```
foreach (NUMBER ::= NUM, PROTOCOL ::= {ip|tcp|udp}:
        access-list NUMBER permit PROTOCOL ...)
{
    <compound(NUMBER, PROTOCOL)>
}
```

In this case, the quantor will bind (NUMBER, PROTOCOL) pairs rather than independent NUMBERS and PROTOCOLS, according to the actual clauses matching *<selector>* pattern 'access-list NUMBER permit PROTOCOL ...'.

- Even if the macro has not been bound previously, the *<selector>* may contain *<macro>* in negated format. For example, the quantor compound

```
foreach (X ::= NAME: define [!X NAME]* X [NAME]*) { <compound> }
```

applied to clause 'define A B C D A E' will bind X to { A, B, C, D, E }. Note that the leading '[!X NAME]*' in the *<selector>* pattern prevents the second occurrence of 'A' in the list to be matched, which is required to prevent ambiguities in matching the selector (cf. Chapter 3.1, Section »Conflicts: Greedy or minimal pattern interpretation?«).

- Macros defined within a quantor compound are strictly local to this compound, that is, there may be more than one quantor, each denoting its variable with 'X', and each assigning a different format to 'X', for example:

```
foreach (X ::= NAME: match address X) { <compound1> } &
foreach (X ::= 100-199: match address X) { <compound2> }
```

In contrast to ordinary macros defined in 'CompoundPatterns.conf', quantor

macro names may be reused for different quantor declarations, so there is no need to invent a new name for every new quantor.

- If there is not a single occurrence of the `<selector>` pattern of a 'foreach' quantor in the IOS configuration, the quantor does never bind the variable(s). Thus, CROCODILE does not need to evaluate the `<compound>` in the body of the quantor pattern, so the body of the quantor never fails. Consequently, the quantor compound is considered 'SATISFIED' (i.e., true=YES) if it occurs as a subcomponent of a more complex compound. However, if the 'foreach' is a top-level quantor compound, then it is considered irrelevant – similar to an IF...THEN compound if the IF-condition is not satisfied — and no annotations are generated.

- The quantor compound

```
exists (<macro>, ..., <macro>: <selector>) { <compound> }
```

is logically equivalent to the quantor compound

```
NOT foreach (<macro>, ..., <macro>: <selector>) { NOT <compound> }
```

The user may freely choose the version that yields the most readable and natural formulation.

Reporting evaluation results

The findings for each compound pattern are output as an annotation. Figure 12 shows a typical excerpt from an evaluation report.

If the user does not assign a message string to the pattern's severity tag(s), a default text is automatically provided. Depending on the severity of the finding and whether a compound pattern matched or failed to match, CROCODILE varies the wording of the output accordingly. Usually, only those pattern components are reported that contributed to the success or failure of a check. Terms within the compound pattern that did not affect the overall result (e.g., non-satisfied operand terms of an OR condition that is satisfied as a whole) are suppressed in the output for better readability. The expression named in the output may therefore differ considerably from the underlying compound as specified in 'CompoundPatterns.conf'. To ensure a unique mapping nevertheless, the line number within the configuration file where the compound was defined is given in brackets at the end of the output line for reference. In interactive mode, CROCODILE provides a corresponding hyperlink pointing directly to this line in 'CompoundPatterns.conf' in the »More Information« section, shown in the lower left of the display. By clicking this link with the mouse the user has conve-

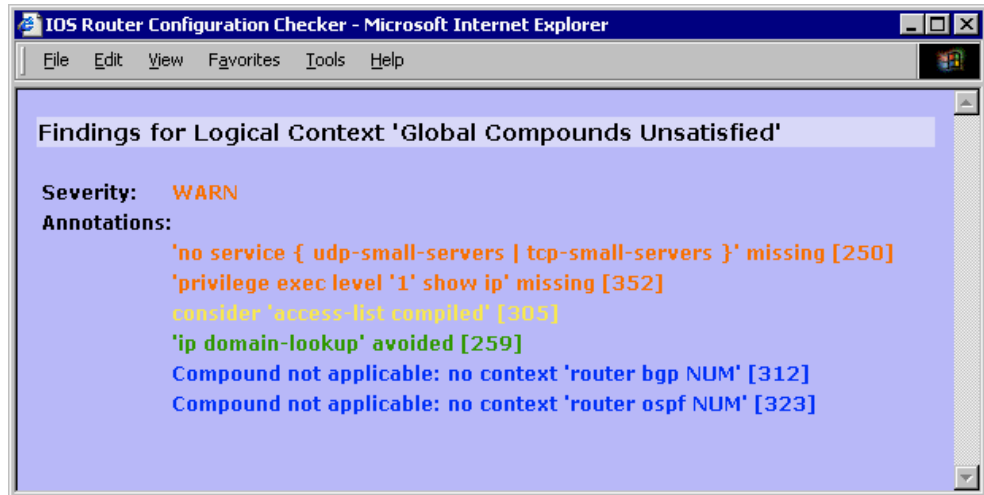


Figure 12 Typical result output of the 'CompoundPatterns' checker module

nient access to the compound rule's definition and its context for further reference.

Avoiding explicit message specifications and relying on default messages is convenient, but appropriate only for the most trivial and self-explanatory compounds. We strongly recommend that a rule author explains the meaning of any non-trivial compound rule by supplying a suitable message string in the configuration file. For further reference, the line number (added in brackets at the end of the message string) and a corresponding hyperlink in the »More Information« section refer the user directly to the compound pattern specification that triggered the message appearing in the evaluation report.

Assigning evaluation results to contexts and bookmarks

By default, all annotations generated by the CompoundPatterns module are assigned either to the configuration clause that caused the compound pattern to match, or — if the pattern cannot be attributed to a specific configuration clause — to the clause context of those configuration clauses that affected the pattern; patterns that refer to global configuration clauses, but cannot be attributed to any specific clause, assign their annotations to the generic logical contexts 'Compounds Omitted' and 'Global Compounds Unsatisfied', respectively.

The user may, however, add more structure to the evaluation results by grouping the annotations generated by compound patterns into more specific logical contexts. For example, all compound patterns referring to access control lists may report their findings to a context named 'Access Control Lists'. To specify the

name assignment, the user may simply put a context directive in front of a sequence of compound pattern specifications in 'CompoundPatterns.conf':

```

SET CONTEXT <context name1>
<compound pattern specifications>
...

SET CONTEXT <context name2>
<compound pattern specifications>
...

UNSET CONTEXT

```

Any annotation caused by subsequent compound pattern specifications will then appear under the given context name in the evaluation report. A 'SET CONTEXT' directive controls the context assignment of all subsequent compound specifications until a new context name is declared, or until the context assignment is revoked by an 'UNSET CONTEXT' directive. All context directives must appear as a separate line of the 'CompoundPatterns.conf' configuration file.

To further improve the structure of the online report, context names declared with 'SET CONTEXT' may be bookmarked by adding a bookmark directive:

```

SET BOOKMARK <bookmark name1>
SET CONTEXT <context name1>
...
SET CONTEXT <context name2>
...

SET BOOKMARK <bookmark name2>
SET CONTEXT <context name3>
...

UNSET BOOKMARK

```

In this example, both <context name1> and <context name2> appear under bookmark <bookmark name1>, while <context name3> will be bookmarked as <bookmark name2>. Using such bookmark and context directives, compound pattern specifications may even direct their compound annotations to bookmarks and contexts that are used by other plugin modules. For example, compounds referring to proper NTP configuration settings may add their annotations to the bookmark 'NTP' that is used also by the 'NTP' plugin module.

If contexts are declared with 'SET CONTEXT' but no bookmark has been declared with a 'SET BOOKMARK' directive, then these contexts are collected under the default bookmark 'CompoundPatterns'.

Remarks

The checker module 'CompoundPatterns' covers a remarkable scope of application. The typical checking tasks suitable for 'CompoundPatterns' include

- Check whether desired router services and modes of operation are explicitly activated, and if undesired services are explicitly deactivated
- Check whether certain commands have been assigned the required level of privilege.
- Check whether banner texts have been specified and are displayed at the appropriate opportunity
- Check whether passwords have been specified and assigned, and whether these passwords are encrypted securely
- Check whether items that are used have been defined, and whether items that are defined are actually used.

In spite of its versatility, the checker module is easy to use. Users do not have to program their own checker modules in order to specify evaluation criteria according to their specific requirements. The BNF-like format of the compound specifications and the option of combining them through logical operators enables rather subtle and precise definitions, in particular conditional and quantified rules. The burden of specifying evaluation rules is further reduced by supplying default messages for those compounds that need no further explanation.

Being simple to configure, the module may also serve as a training device for users intending to develop new checker modules. They can use 'CompoundPatterns.conf' to gain experience with the pattern and macro syntax before attempting to program their own pattern handlers. Pattern specifications may be tested with little effort using 'CompoundPatterns' before they serve as triggers for a new checker module.

The price for the versatility and simplicity of the rule syntax is the module's limited reaction when a pattern match is found. The user cannot specify a flexible action beyond generating a specific message.

4.3 Module 'IngressEgress'

Routers positioned at the border of a network that are intended to handle the connection to other networks (so-called perimeter routers) should check traffic

crossing the inbound and outbound interfaces for plausibility. In inbound direction (ingress), they protect their local network from unwanted, erroneous, or even malicious data packets; in outbound direction (egress), they prevent confusion or even manipulation attempts originating from their local network.

IOS routers usually realize ingress and egress filtering by means of suitable access control lists. The 'IngressEgress' checker module supports the exact specification of desired filtering effects and checks whether an ACL indeed obeys this specification.

The 'IngressEgress' checker module builds on evaluation results of the 'Connectivity' checker module (see Chapter 4.5, page 66 ff) and may therefore only be used in conjunction with the latter. If 'Connectivity' is not included in an evaluation run, 'IngressEgress' cannot run either. The tool checks this dependency and automatically loads the partner module 'Connectivity' if needed.

Evaluation criteria

The evaluation criteria of the 'IngressEgress' checker module may be freely configured. The module reads its specific evaluation rules from the configuration file 'IngressEgress.conf' located in the 'Configuration' subdirectory (see directory structure in Figure 10, page 39).

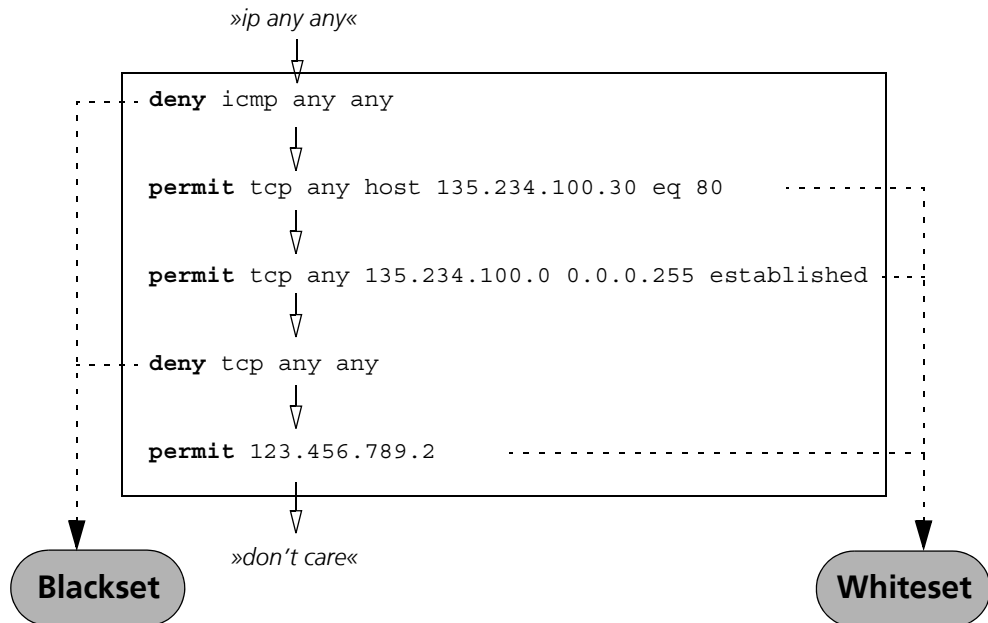
The configuration file contains requirements for the ingress/egress filtering for

- interfaces,
- line interfaces, or
- access control lists.

The user may specify the whiteset and the blackset for these. *Whitesets* determine which types of packets must *at least* be accepted by an interface, line interface, or ACL. *Blacksets* determine which types of packet must *at least* be rejected. If a packet falls neither into the whiteset nor the blackset, it may be accepted or rejected by this router configuration — the tool will allow both options.

Principle of filter specification

Syntax and semantics of ingress/egress filtering specifications mirror those of IOS commands. This offers the user a familiar, easily mastered command format.

**Figure 13**

Specification of ingress/egress settings: Whiteset and Blackset

The principle of description is simple: The user specifies a »filter list« mirroring the desired filtering effect. The checker module computes the corresponding whiteset (set of all accepted packets) and blackset (set of all rejected packets). Later in the evaluation, the tool checks whether all elements of the whiteset are actually accepted by the ACL or (line) interface, and whether all elements of the blackset are actually rejected. Parts of the whiteset that are falsely rejected and parts of the blackset that are erroneously accepted are listed as warnings (Figure 14 on page 62 gives an example of a typical error report).

Figure 13 shows a sample configuration and the principle of computing its whiteset and blackset. A specification consists of a sequence of 'deny' and 'permit' clauses. They are applied in order, from top to bottom. In the case of a 'deny' clause, the specified subset is subtracted from the set of all possible packets (»ip any any«) and added to the blackset. According to Figure 13, for instance, all ICMP packets are added to the blackset. Each subsequent clause is applied to the remaining set of packets in the same way. In Figure 13, the next clause is a 'permit'.

Each 'permit' clause branches the corresponding subset of packets off to the whiteset. In the example, these are all TCP packets with destination address 135.234.100.30, port 80. All other packets — in this example, all except ICMP packets and TCP packets to the indicated port — remain and will be passed on to the next filtering clause. Clause after clause is processed in this way.

With this method a pure whiteset specification results by employing only 'permit' clauses. If the filter is described using only 'deny' clauses, a pure blackset specification results. Mixed forms as in Figure 13 are possible as well: CROCODILE will automatically separate the whitelist from the blacklist part of the specification, taking into account the order of the specified clauses.

The interpretation of the filtering specifications thus corresponds largely to the way IOS evaluates ACLs. Most users will be familiar with this mechanism. However, in contrast to ACLs, there is no implicit »deny any« at the end of an 'IngressEgress' filtering specification. Packets left over after the final clause and not assigned to either the whiteset or blackset remain undecided instead: The router may arbitrarily accept or reject them. In an ACL, on the other hand, all remaining packets will be rejected implicitly by the router. If a filtering specification is to achieve exactly the same semantics as an ACL, an explicit »deny any« clause must be added at the end of the specification.

The process illustrated in Figure 13 results in two sets of packets: The whiteset contains (conceptually) all conceivable packets that should be allowed according to the filtering specification; the blackset comprises (conceptually) all conceivable packets that are to be rejected according to the filtering specification. The 'IngressEgress' is able to check for each of these packets whether it is indeed accepted or rejected when running this router configuration.¹² Each deviation is recorded exactly.

Figure 14 shows a typical excerpt from the result of an 'IngressEgress' evaluation. In this example, a whiteset and blackset specification for the access to line interfaces of the form »vty ...« is checked in inbound direction (»in«). The figure shows the specification in its top half, including the evaluation result for the line interface 'vty 0', which is protected by the standard ACL no. 2 according to this specification. It can be seen that ACL 2 is too restrictive regarding the 'telnet' port, but too liberal with respect to certain packet origins.

The specification method sketched above works independently of the order or the format of the filtering clauses: Only the effect resulting from the interplay of all clauses is relevant. In particular, this means that the specified filtering clauses do not necessarily have to appear literally in the router configuration. It is sufficient if the overall configuration amounts to the desired effect, namely, accepting the whiteset and rejecting the blackset — using whatever ACL constructions! This clearly distinguishes the 'IngressEgress' checker module from other simple

¹² This description refers to an abstract concept: The technical implementation certainly refrains from actually constructing the described sets element by element. Such a naive approach would fail due to the enormous cardinality of the sets. Instead, whitesets and blacksets are described symbolically only, and they are analyzed using transformation steps at the symbolic level.

```

Ingress/Egress for Line (in):
-----

PATTERN = 'vty ...'
Whiteset = {
tcp 153.96.133.* *.*.*.* eq telnet
}
Blackset = {
ip [0-152,154-255].*.*.* *.*.*.*
ip 153.[0-95,97-255].*.* *.*.*.*
ip 153.96.[0-132,134-255].* *.*.*.*
[0-5,7-255] 153.96.133.* *.*.*.*
tcp 153.96.133.* *.*.*.* [0-22,24-65535]
}

Line 'vty 0'
Accesslist: 2
--> Whiteset only partially supported. Missing:
tcp 153.96.133.* *.*.*.* eq telnet
--> Blackset only partially supported. Missing:
ip 151.200.19.52 *.*.*.*
ip 164.47.133.2 *.*.*.*
ip 217.31.34.129 *.*.*.*

```

Figure 14

Typical example for the result of an 'IngressEgress' evaluation

checking scripts available on the Internet that merely check the specification and the configuration for syntactical equivalence.

Available 'permit' and 'deny' clauses

Filtering clauses follow the IOS format of standard access control lists and extended access control lists. The syntax for valid filtering clauses (in the standard format) is

```
{permit|deny} {any | IPADDR [WILDCARDMASK]}
```

or (in extended format)

```
{permit|deny} PROTOCOL SUBNET [PORTSPEC] SUBNET [PORTSPEC|ICMPSPEC]
[established]
```

with

```

PROTOCOL ::= {0-255 | <protocol name>}
SUBNET    ::= {IPADDR WILDCARDMASK | any | host IPADDR}
PORTSPEC  ::= {{lt|gt|eq|neq} <port> | range <Port> <Port>}

```

```
ICMPSPEC ::= {0-255 [0-255] | <type name>}
```

This corresponds to the available parameter settings and effect of corresponding IOS commands.

In contrast to IOS configurations, standard clauses may freely be mixed with extended clauses in an 'IngressEgress' filter specification. Specifications in extended format may also be used to check standard ACLs — and vice versa.

As mentioned in the beginning, filtering specifications may be assigned to certain ACLs or certain interfaces. This may be used to create differing filtering specifications for the different router access points.

Evaluation criteria for access control lists

In the simplest case, the user specifies criteria for the filtering effect of ACLs. It is the user's responsibility then to assign these ACLs to interfaces appropriately.

The specification format for 'access-list' criteria is as follows:

```
access-list <pattern> {  
  <list of 'permit' and 'deny' filtering clauses>  
}
```

The pattern is a BNF-like syntax description (corresponding to the pattern format for handlers, see Chapter 3.2). It determines which ACLs are subject to this specification. Only those lists whose name matches the pattern are affected. For example, if "[1-99]" is used as a pattern, the specification applies to all standard ACLs; the pattern "...", on the other hand, makes the specification apply to all ACLs. Arbitrarily complex patterns may be specified.

Evaluation criteria for interfaces

The user may also assign a filtering specification directly to certain interfaces. The checker module will then analyze these interfaces, taking into account the ACLs they are protected by.

The specification format for such »interface« rules is similar to the above; however, for an interface the intended direction of the specification must be indicated, using "in" for inbound direction, "out" for outbound, and "inout" for both directions. The syntax is:

```
interface <Pattern> {in|out|inout} {  
  <list of 'permit' and 'deny' filtering clauses>  
}
```

```

! RFC1918 inbound address filtering

interface { Ethernet | Serial } ... in {
    deny    ip 10.0.0.0 0.255.255.255 any
    deny    ip 127.0.0.0 0.255.255.255 any
    deny    ip 172.16.0.0 0.15.255.255 any
    deny    ip 192.168.0.0 0.0.255.255 any
}

! RFC1918 outbound address filtering

interface { Ethernet | Serial } ... out {
    deny    ip any 10.0.0.0 0.255.255.255
    deny    ip any 127.0.0.0 0.255.255.255
    deny    ip any 172.16.0.0 0.15.255.255
    deny    ip any 192.168.0.0 0.0.255.255
}

```

Figure 15

Example of an 'IngressEgress' filtering specification for interfaces

Figure 15 shows an example for an interface specification along the recommendations for filtering non-public IP address ranges according to [RFC1918]. The rules apply to all 'Serial' and 'Ethernet' interfaces. (Note that lines starting with '!' or '#' are treated as comments — as is traditional under IOS and Unix — and are ignored by the tool.)

Evaluation criteria for line interfaces

Line interfaces may be equipped with filtering specifications like other interfaces. The specification format is analogous:

```

line <pattern> {in|out|inout} {
    <list of 'permit' and 'deny' filtering clauses>
}

```

Using macros

When formulating filtering specifications, recurring text sequences or specification building blocks are common. For the sake of readability and maintainability, such building blocks should be assigned symbolic names. To this end, the checker module supports macros in 'IngressEgress.conf'. The format is as follows:

```

<MACRO NAME> ::= <pattern>

```

A macro declaration may comprise only one line, and the macro name may contain only capital letters. Any BNF pattern may be used, provided it is

```

NETMASK    ::= 0.0.0.255
LOCALNETA  ::= 147.56.103.0 NETMASK
LOCALNETB  ::= 147.56.104.0 NETMASK
LOCALNETC  ::= 147.56.105.0 NETMASK

OUTBOUND   ::= Serial [0-1]

interface OUTBOUND out {
    deny    ip any 10.0.0.0 0.255.255.255
    deny    ip any 127.0.0.0 0.255.255.255
    deny    ip any 172.16.0.0 0.15.255.255
    deny    ip any 192.168.0.0 0.0.255.255
    permit  ip LOCALNETA any
    permit  ip LOCALNETB any
    permit  ip LOCALNETC any
}

```

Figure 16

Examples for the use of macros in 'IngressEgress.conf'

compatible with the intended use of the macro. Figure 16 shows some examples for the use of macros.

Remarks

The special »charm« of the 'IngressEgress' module resides in its simple and flexible configuration on the one hand, and in its ability to treat whitesets and blacksets completely at the logical level without resorting to the way they were constructed on the other hand. That is, even if the filtering specification shows no syntactic similarity with the relevant parts of the router configuration, the checker module recognizes nevertheless whether both descriptions are logically equivalent — a unique ability as far as we know.

This checker module is remarkable from a technical perspective as well. In order to evaluate its configuration file 'IngressEgress.conf', it leverages the existing functionality of the parser and pattern matcher. This enables the recognition of specification clauses of the most diverse form; further, typing errors in the specification may be localized close to the exact character where the error occurs and be exactly reported. Using standard BNF patterns in the definition of the specification format also provides the option to modify and extend the syntax format of certain configuration clauses on demand and with little effort.

4.4 Module 'Routing'

The 'Routing' checker module analyzes the (statically defined) relation between IP subnets and IP interfaces and determines for each interface the subnets that are reachable via this route — and vice versa. This analysis provides information

about the network neighborhood of the router that is used by module 'Connectivity' to construct a connectivity chart (see Chapter 4.5).

Evaluation criteria

The 'Routing' module has no corresponding configuration file. All evaluation criteria are hard-coded within the module. It collects the following information:

- Which IP subnets are directly connected to an interface, that is, which IP addresses and subnet masks are assigned to each interface?
- Which static IP routes are defined, that is, which subnets are routed to specific interfaces or destination addresses? What is the »distance« assigned to each route.

While analyzing the corresponding IOS clauses, the module checks for inconsistencies such as circular routes, or routes to undefined interfaces. All findings are grouped together in the logical context 'Routing'.

Based on static routing information, the module generates two lists: »Interfaces to Routes« and »Routes to Interfaces«. These lists are accessible via bookmark 'Connectivity and Routing'. They summarize how subnets can be reached via interfaces and vice versa, with the specified »distance« of the route in square brackets. Figure 17 shows a typical result. In this example, lines 104 and 105 of the IOS router configuration define a loop, and therefore address ranges 10.*.* and 172.16.*.* are caught in a circular route.

Remark

Module 'Connectivity' relies on information provided by module 'Routing'. Whenever 'Connectivity' is selected in the configuration file 'plugin.conf', 'Routing' is automatically plugged in as a »REQUIRED_MODULE«, too.

4.5 Module 'Connectivity'

The 'Connectivity' checker module reconstructs the network topology around the router with its surrounding subnets and nodes and creates a graphical connectivity chart. Furthermore, it gathers all information about the allowed data flows between the router interfaces, about routes and potentially involved communication partners, about the employed communication protocols and installed filtering rules. This data is partly included in the connectivity chart, and partly listed in textual form. Additionally, information about access control lists

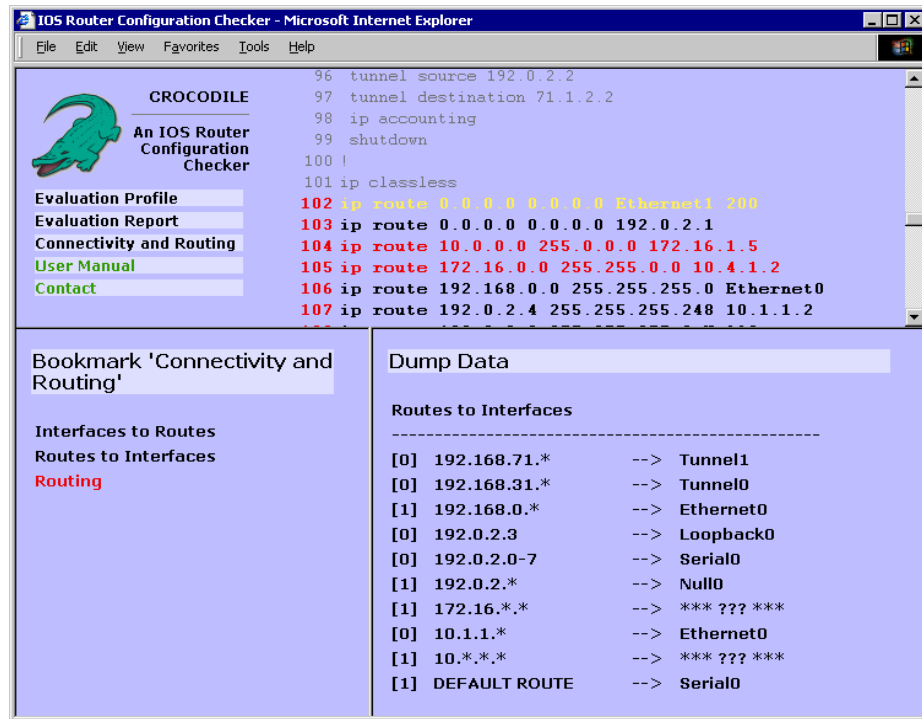


Figure 17 Typical evaluation result of module 'Routing'

and reachability is stored in XML format for postprocessing with CROCODILE's 'blackwhite' utility (see Chapter 7.1).

Figure 18 shows a connectivity chart that was automatically generated by 'Connectivity' based on a router's static configuration attributes. In this chart, directly connected subnets (distance = 0) are connected with a red, doubled line. Disabled interfaces are shown in gray color, and the name of the default interface is printed in white. If CROCODILE can derive from the IOS configuration that a subnet contains »interesting« nodes (such as tunnel destinations, logging servers, web servers), an annotated IP addresses of these nodes appears in the corresponding subnet box.

Evaluation criteria

The 'Connectivity' module has no corresponding configuration file. All evaluation criteria are hard-coded within the module.

The checker module only marginally contributes to direct security analysis. Its primary task is to gather an overview of the network connections of the router. As far as curiosities and irregularities surface during this task, they are certainly

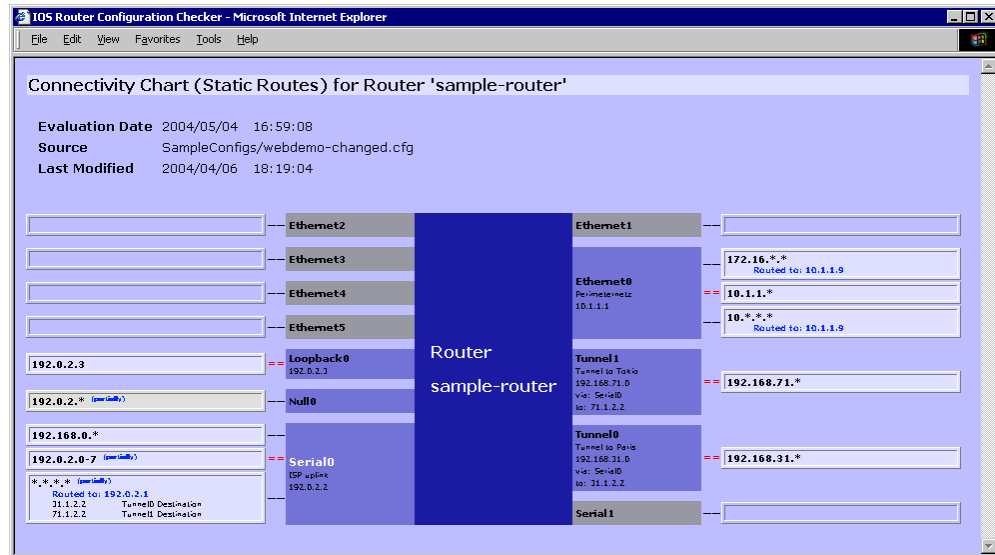


Figure 18 Example of a connectivity chart generated with 'Connectivity'

reported. However, the focus is only on information gathering and processing to aid subsequent analysis — performed manually, or by other checker modules. The following information is gathered jointly by 'Connectivity' in cooperation with the 'Routing' module for this purpose:

Interfaces

For all IP interfaces, the essential data about the interface and the nodes and subnets reachable through this interface are recorded, including:

- IP and subnet address of the interface as well as its IP broadcast address
- clear text description (as far as available in the router configuration)
- activation state
- subnets and nodes reachable through the interface, as well as the protocols to access these nodes (as far as implied by the router configuration)
- installed ACLs for inbound traffic and the resulting protocol connections accepted
- installed filter list for outbound traffic and the propagated protocol connections permitted by this

```

Interface ethernet 5
-----
Description:      LAN SIGMA administration network
Enable state:     (implicitly) enabled
Address(es):      172.30.128.1
AccessGrp OUT:    105
Subnets behind this interface:
                  172.30.128.*
                  Node 172.30.128.45   ip
                  Node 172.30.128.2    ip

Accepted connections:
ip *.*.*.* *.*.*.*

Propagated connections:
[0,2-255] 153.96.133-135.* 172.30.128.*
[0,2-255] 172.30.1.* 172.30.128.[2,45]
[0,2-255] 172.30.200.11 172.30.128.*
[0,2-255] 172.30.200.2 172.30.128.2
[0,2-255] 172.30.200.5 172.30.128.45
icmp *.*.*.* 172.30.128.*

```

Figure 19

Typical information of the 'Connectivity' checker module regarding an interface

- line numbers within the configuration description where the interface is referenced

Figure 19 shows a typical output of the checker module regarding interface attributes. Most remarkable are the sections »Accepted Connections« and »Propagated Connections«. In highly condensed form, they indicate which IP protocols (from the protocol number range 0–255) are propagated as inbound and outbound traffic by the interface. In order to compute this, various configuration clauses must be evaluated, including interface declarations, ACLs and static routes. The two lines

```

[0,2-255] 172.30.1.* 172.30.128.[2,45]
icmp *.*.*.* 172.30.128.*

```

(cf. Figure 19) for example must be read as follows:

- All packets originating from the IP address '172.30.1.*' and destined to the IP address '172.30.128.2' or '172.30.128.45' may pass, provided that their protocol number is not equal to 1 — that is, either 0 or 2-255. (Protocol number 1 designates the ICMP protocol.)
- For the ICMP protocol previously exempted the following holds: ICMP packets are legal from any origin to all destinations within the '172.30.128.*' subnet.

The remaining lines of the »Accepted Connections« and »Propagated Connections« may be read accordingly.

Line Interfaces

The relevant attributes of the router's administration interfaces — so-called line interfaces — are recorded in a format similar to that of the other interfaces. Figure 20 shows a typical example.

Tunnel Interfaces

The module tries to determine the physical routes of tunnel connections, that is, the interface over which the tunnel connections leave the router. Of course, this analysis is based on static routing information only. Dynamic routing protocols may change the routes that are actually chosen in operation, but these are not considered here. The table of physical tunnel routes is accessible via the 'Connectivity and Routing' bookmark.

Subnets

The checker module derives the address ranges of the subnets that are explicitly mentioned in the router configuration description, either in the context of interfaces addresses, when defining static routes, or as origins or destinations of ACL clauses. Considering that a security auditor is normally aware of the addresses of the most important surrounding subnets, a systematic report is helpful to assign the individual router interfaces to certain functional areas. Subnet information is shown in the connectivity chart (Figure 18), which can be selected via the 'Connectivity and Routing' bookmark.

```

Line console 0
-----
AccessGrp IN: 1

Accepted connections:
ip 151.200.19.52 *.*.*.*
ip 164.47.133.2  *.*.*.*
ip 217.31.34.129 *.*.*.*

Propagated connections:
ip *.*.*.* *.*.*.*

```

Figure 20

Typical information by the 'Connectivity' module regarding a line interface

```

Node 217.31.32.81
-----
Located behind:  ethernet 1
Ports:          www smtp syslog
Protocols:      tcp udp icmp

```

Figure 21

Typical information provided by the 'Connectivity' module regarding network nodes

Nodes

Individual IP addresses explicitly mentioned within the router configuration usually designate network nodes — in most cases, these are computers or IP routers. The checker module lists all »interesting« IP nodes and delivers the following information about them:

- IP address of the node and subnet containing this node
- interface by which this node may be reached
- IP protocols and ports used to access this node, or function of the node, respectively

The evaluation results are partly shown in the connectivity chart, partly listed in text format. Figure 21 shows a typical module output. In this case, the ports suggests that the node is a WWW, mail, and logging server.

Access control lists

Access [control] lists (ACLs) are among the hardest router configuration features to analyze. The checker module tries to clarify the filtering effect of such lists by transforming them to a whitelist format (i.e., the exhaustive list of all allowed connections). Apart from that, 'Connectivity' checks which relationships hold between single ACL clauses and whether an ACL contains clauses that have no effect — so-called »dead« clauses. The gathered data include the following:

- type of the ACL (standard or extended)
- correct use (typo protection!):
 - each defined ACL is used somewhere
 - each used ACL is defined somewhere

- explicit reset (i.e., deletion) of the ACL before any definitions?
- interplay of different rules within one ACL

The evaluation results are stored as annotations in the result data base. These annotations directly linked to the corresponding configuration clauses in the output display.

Advanced analyses of ACLs or interfaces may be performed using the 'black-white' utility (see Chapter 7.1). The 'Connectivity' module provides the relevant raw data for this utility in XML format.

Remark

Module 'Connectivity' requires data gathered by module 'Routing'. Thus, whenever 'Connectivity' is plugged in, CROCODILE automatically plugs in 'Routing', too.

4.6 Module 'SNMP'

The SNMP module checks the configuration settings of the router regarding the Simple Network Management Protocol (SNMP). SNMP supports querying and modifying relevant state information in the Management Information Base (MIB) of network components, and reporting exceptional events (traps) in the network to a central management station. It is an indispensable de facto standard for network management.

SNMP provides security-critical functionality, as it enables spying or even reconfiguring the router from a remote management node. Therefore, SNMP settings require particular care.

Evaluation criteria

The 'SNMP' checker module has no configurable checking rules. All evaluation criteria are hard-coded within the module.

The evaluation is complicated by the three different SNMP versions supported by IOS that differ regarding their concepts for authentication and authorization. The checker module evaluates SNMPv1, v2 and v3 clauses, and even a combination of clauses referring to different versions, if absolutely required.

The user is basically free to declare a mixed SNMP configuration supporting different versions at the same time. However, this may cause effects that are hard

to predict and may only be fully understood by experimentation.¹³ Due to missing technical documentation the checker module is not able to compute comprehensive predictions of the router's behavior in mixed SNMP mode. It only produces a rough analysis of the correctness of SNMP clauses in such cases.

Authentication, authorization and functional scope

SNMP provides various mechanisms to alleviate the potential for misuse, in particular in SNMPv3. Router configurations should use the full spectrum of options for authentication and authorization in order to restrict access to the router in the best possible way. Dispensable functionality should best be turned off completely.

Regarding traps — spontaneous notifications sent by the router about exceptional conditions of operation — it is advisable not to deactivate important messages. Conversely, frequent and less relevant events should be signalled only sparingly and thus be mostly deactivated.

The checker module therefore primarily analyzes the following issues:

- Is the SNMP server function deactivated?
- Are community strings («SNMP passwords») sufficiently hard to guess?
- Are the available authentication and authorization options put to best use?
- Is the availability of SNMP information restricted by views?
- Is access restricted to READONLY where possible?
- Is SNMP access restricted by access control lists?
- Is a router shutdown via SNMP excluded?
- Which SNMP traps are activated, and which are deactivated?

¹³ In our own experiments, we observed a sometimes rather weird behavior of routers with mixed configurations. It remained unclear whether this was due to an accidental property of that type of router or to a reliable feature of the employed IOS version. Because of such experiences, we advise against mixed configurations.

Manager settings

Routers usually belong to the managed objects, not the managing ones. It is thus unusual to configure a router as an SNMP manager. The module warns if the router is configured as a manager, able to reconfigure other SNMP components.

Miscellaneous

On the whole, the available IOS clauses for SNMP configuration are rather intricate and riddled with side effects that are only very vaguely documented. In order to avoid problems, following the recommendations by Cisco is advised. The tools thus checks the following, among other things:

- Are SNMP clauses declared in the order recommended by Cisco?
- Are any deprecated or useless (i.e., overridden) clauses presents?
- Are any unusual packet sizes used?

Remark

From a technical point of view, it is perfectly conceivable to improve the evaluation depth of the SNMP module considerably. At present, this is mostly hampered by the insufficient documentation from Cisco regarding SNMP configuration under IOS. In particular, it seems desirable to model the exactly interplay of SNMP v1, v2, and v3 in the NTP module, in order to better predict mixed configurations.

4.7 Module 'NTP'

The NTP checker module evaluates the configuration settings of the router regarding the Network Time Protocol (NTP). NTP synchronizes the local clocks of network components. Synchronous clocks are an important aid for monitoring and error correction in communication networks.

The NTP module builds on analyses by the 'Connectivity' module (cf. Chapter 4.5) and may therefore only be activated in conjunction with that module. If 'Connectivity' is not explicitly included in an evaluation run, it will be automatically loaded for the dependent module 'NTP'.

Evaluation criteria

The NTP module has no configurable checking rules. All evaluation criteria are hard-coded within the module's source code.

The checker module identifies possible weaknesses and errors in the NTP configuration. It points out contradictory NTP configuration clauses and the omission of certain standard settings.

Apart from that, it compiles a list that easily shows the host this router receives its system time from and the hosts it may pass it on to. The option of sending NTP messages as broadcasts is also taken into account.

Contradictions and logical errors

While checking the lines of the configuration description, the module judges the consistency of the commands. It checks whether all employed keys and ACLs are actually defined. Conversely, it checks whether all declared keys and ACLs are actually referenced. Redefining or deleting a previously declared key is not necessary in a ready-made script, but it may indicate a hasty local change of the router configuration — a potential error.

Further, missing configuration parameters are also flagged during this phase. If keys or ACL commands contain default values, this is reported and used in the subsequent analysis. If certain commands are not allowed in a specific IOS mode, this will be reported.

Many errors result only from the logical interplay of all configuration clauses. For instance, if authentication by keys has been configured, the 'ntp-server' clause has no effect without declaring a key. The module tries to find such irregularities.

List of IP addresses and interfaces

After an evaluation run, the IP addresses of all servers and peers involved in NTP communication with the router are listed. Further, the interfaces for incoming and outgoing NTP broadcast messages are reported. This information has already been checked for plausibility. In addition to that, the ACLs employed using 'ntp access-group' are analyzed. However, the network administrator has to check the ACLs of the involved interfaces: Traffic to port 123 (TCP or UDP) must be allowed through at least one interface to prevent discarding the NTP messages due to such a restriction. An automatic control is not possible here, as the tool has no knowledge of the network neighborhood and thus cannot decide *which* interface should be allowed for NTP.

Directly attaching a clock to the router is also covered. Additionally, a list of the possible and the trustworthy keys is displayed if authentication by keys has been chosen. In this case, the keys must be made known to an NTP server in order to make the router actually accept incoming messages and set its own clock accordingly. Thus, the network administrator should review the list of all peers and servers, and he should check the keys configured on these machines. The lists of locally declared and used authentication keys are very helpful for this check.

The router as an NTP server

A Cisco router will normally answer NTP requests with its software timer. However, if the router has not been configured by another time source before, the NTP message will have stratum 0. This indicates that the time is not synchronized; a client will ignore this message. If the router has been synchronized by NTP or has been declared a master using the 'ntp-master' command, it will function as an NTP server on all interfaces not explicitly exempted from this setting. The software clock of the router is sent as the time then. For each interface, the checker module will compile the list all hosts potentially sending time requests, based on the ACLs of the interfaces or the global NTP ACL.

Following the bookmark "NTP", a list for each interface will show which hosts may use the router as an NTP server under the current configuration. Of course all router ACLs are also observed here, however, the system administrator must once again check manually whether port 123 (TCP or UDP) is allowed at the interfaces of the involved network components, as the NTP reply messages could otherwise not leave the router. As this requires knowledge of the network neighborhood, it cannot be checked automatically.

Successfully using the router as an NTP server also requires comparing the authentication keys, as the server needs to know the client key. Again, this requires context information not contained in the router configuration.

Logical contexts to improve clarity

The NTP module will also create a separate context gathering all commands referring to the time service. The context display will show only these lines, independent of their distribution across the configuration description. The trained user may thus quickly see the effects of all NTP commands. Another logical context gathers all commands referring to authentication keys (if applicable).

Remarks

NTP normally influences only the software clock of the router ('clock'). However, if the router also has a hardware clock ('calendar'), this may be synchronized by NTP as well (by the 'ntp update-calendar' command). As the hardware time is automatically copied from the calendar to the clock when booting the router, this has the advantage of providing the correct time even after a reboot.

If the calendar is not synchronized by NTP, a considerable drift between calendar and clock may result after a certain time of operation. This is not a problem while the router is running. However, after a reboot the software clock may copy a rather imprecise calendar time. As a host using NTP will set its clock only if the local time does not differ too much from the NTP time, this may have the effect that the router will no longer be synchronized after a reboot and thus run out of the NTP time context without further notice.

For this reason, it is advisable to check the clock of a router after a reboot or an NTP activation using 'show clock' and set the clock manually to the correct time as appropriate.

4.8 Module 'AAA'

The AAA functions control authentication, authorization, and accounting. The 'AAA' checker module analyzes the consistency, completeness, and security of the AAA configuration.

Evaluation criteria

All AAA evaluation criteria are hard-coded in the checker module. There are no configurable checking rules.

At present, the module analyzes only authentication and authorization features. The third issue — accounting — is ignored, as it is not relevant to security. The primary evaluation criteria include:

- Are AAA clauses given in the wrong order?
- Is the configuration of the employed AAA functions incomplete or inconsistent?
- Are there any AAA features that are used but never defined, or defined but never used?
- Are there any insecure AAA configuration settings?

- Are there any IOS settings that are incompatible with AAA?
- Are there any deprecated authentication or authorization settings?

Authentication

Using AAA, various activities may be authorized, including login, dial-in («PPP») or changing into the so-called privileged mode («enable»). For any of these activities, a list of authentication methods may be configured that are tried in order to verify the identity of the user. The methods may optionally use line passwords, the local user database of the router, or external authentication servers of type RADIUS, TACACS+ or Kerberos. Even a null authentication may be allowed as a fallback.

The module checks whether the declared method lists are actually used and whether they are acceptable from a security perspective. If line passwords or the user database are chosen, the module checks whether corresponding passwords or password-protected local accounts are actually configured. If an external server is employed, the module checks whether such a server has actually been configured.

If certain activities require a list of authentication methods, such a list must be defined. The checker module ensures that no empty or undefined method lists are used by mischance.

Authorization

AAA supports subjecting certain groups of IOS commands to a separate authorization. Authorization may be required, for instance, for dial-in, shell access, reverse telnet, or for a so-called privilege level. Similar to authentication, authorization method lists are defined that serve to check the right to use certain groups of commands. Local authorization according to the user database is supported as well as server-supported authorization by means of RADIUS, TACACS+ or Kerberos.

The evaluation tool ensures that all resources needed to use a certain authorization method are locally configured. It also ensures that all employed method lists are defined and that all defined lists are actually used. The internal consistency of the method list is checked, too.

External authentication and authorization servers

If authentication or authorization rely on an external server, this server must be configured correspondingly. The checker module analyzes whether RADIUS, TACACS+ or Kerberos are needed. If such a server is required, the module checks whether a secure communication using the RADIUS, TACACS or Kerberos protocol is ensured. The relevant attributes to configure include keys for server authentication and uniform origin addresses.

As the communication with an authentication or authorization server may be unavailable from time to time, it is recommended to configure several servers of the same type that may be contacted alternatively. To this end, server groups may be configured. The checker module analyzes whether all group names that are in use are defined and whether all servers of a group have been configured.

When a TACACS server is used, AAA assumes the TACACS+ protocol. Older protocol variants — TACACS or Extended TACACS — are not compatible with AAA. The checker module warns if AAA has been configured in combination with one of the deprecated protocols.

Remarks

The 'AAA' module is one of the most complex checker modules so far. It comprises more than 1600 lines of Perl code. One reason for this is that authentication and authorization is a complex issue.¹⁴

Further, it appears that the AAA mechanisms are not very mature yet and subject to constant change. Cisco has changed the syntax and function several times over a few IOS generations. Obvious errors and contradictions have entered the AAA documentation during these modifications, and the conceptual consistency of the AAA directives has suffered considerably.

Checking the AAA functionality is necessarily incomplete if external servers are involved. In this case, the security depends on the server configuration as well as the router configuration. Thus, apart from analyzing the routers, the employed servers must be included in the analysis. The tool cannot perform this.

¹⁴ This already shows by the 30 patterns required to cover the configuration clauses relevant to AAA. For the analysis, interface, user, line, and server declarations must be taken into account, too.

4.9 Module 'Logging'

Cisco IOS generates different types of alerts, informational and debug messages about relevant hardware and software events in the router. These messages may be directed to a console terminal, an internal log buffer, or to an external logging server. These logs record an event history of the router operation — an important building block for debugging as well as for a comprehensive IT security concept.

The checker module 'Logging' evaluates the configuration settings applying to logging for correctness, consistency, and completeness. It creates an ordered overview presentation of all relevant parameters.

Evaluation criteria

All evaluation criteria regarding 'Logging' are hard-coded in the source code of the checker module; there are no configurable checking rules.

IOS offers five different ways of log data capturing: three local variants and two network-based ones. Log messages may be sent

- to the router console
- to a terminal line interface (monitor)
- to a local buffer memory
- to a remote SYSLOG server
- to a remote SNMP server

To monitor the router operation systematically, the last three variants in particular are recommended¹⁵; they are analyzed with particular care. All five logging options may be used independently and in parallel. For each variant, the following issues are checked:

- Is the logging option activated, and are all configuration conditions for a proper logging given?

¹⁵ Directly logging to an output device is of only secondary interest for several reasons: First, a subsequent processing with text processing tools is hardly possible any more. Second, the display area of a console may quickly overflow, leading to loss of data. And third, the logging messages are interspersed with user input and other IOS output. All of this complicates the analysis.

- From what level of severity are messages logged? IOS discriminates the following levels of severity in increasing order of importance: debugging, informational, notifications, warnings, errors, critical, alerts, emergencies.
- Is the maximum allowed logging rate limited?
- In the case of SYSLOG: Under what facility name are the messages transmitted to the server?

Apart from that, various general issues are analyzed:

- Are all messages time-stamped?
- How are the logging buffers sized?
- Are additional logging services activated (e.g., line card logging, VIP card logging, BGP or EIGRP Logging, MPLS Logging)?
- Are ACLs monitored using the 'log' attribute, and is the severity level of such messages actually honored for logging?
- Are all configuration clauses syntactically correct, and do all their parameters fall within the legal range of values?
- Are there any contradictory configuration clauses?

SNMP logging

Log messages may be transmitted to an SNMP server using so-called SNMP traps. In order to correctly transmit and receive such traps, several configuration settings must be performed, regarding both logging and SNMP.

The 'Logging' checker module is more or less limited to checking the settings regarding logging. The configuration settings on the SNMP side are mostly ignored. The user may check them using the 'SNMP' checker module (cf. Chapter 4.6). The 'Logging' module refrains from dedicated SNMP rules in order not to duplicate that functionality.

Result processing and reporting

The different evaluation topics are grouped to logical contexts. These contexts are gathered under the 'Logging' bookmark.

Findings about console and terminal logging are mostly reported as neutral (i.e., in the 'INFO' category), unless they constitute definite errors. If the corresponding protocol variant is deactivated, this is not interpreted as worrisome. In contrast to that, the checker module assumes that the local log data buffer as well as an external log data recording (SYSLOG or SNMP) are activated for the sake of security. If there are no corresponding configuration clauses, this result is reported at least in the 'CHECK' category.

Configuration clauses selecting buffer sizes and transmission rates are normally labeled as 'CHECK', independent of the employed attribute values. It is up to the user to judge the adequacy of these attribute values — sensible settings are very application-specific and thus not amenable to an automatic evaluation.

Remark

The checker module purposely refrains from configurable target settings. The evaluation rather concentrates on the internal consistency of the logging configuration and reports the existing settings in a clear manner. This should form a sufficient basis to judge the correctness and security status of the configuration.

4.10 Module 'NAT'

Cisco IOS provides mechanisms for network address translation (NAT). The 'NAT' checker module examines NAT-related configuration clauses for omissions, contradictions, and ambiguities.

Evaluation criteria

All evaluation criteria regarding 'NAT' are hard-coded in the source code of the checker module; there are no configurable checking rules. Among others, the module checks for

- undefined or unreferenced NAT address pools,
- undefined NAT route-maps,
- missing 'inside' or 'outside' NAT interface declarations,
- IP addresses subject to NAT behind interfaces that are not NAT-enabled,
- ambiguous static NAT specifications, and
- static NAT specifications in conflict with dynamic NAT specifications.

In addition to these consistency and correctness checks, the 'NAT' module provides information about static NAT to the 'Connectivity' module; this information is presented in the 'Connectivity' and the 'Nodes' displays: Source address translations appear as 'NAT' entries while destination address translations appear as 'dNAT' entries.

Remark

Module 'NAT' builds on ACL analysis carried out by the 'Connectivity' checker module. Therefore, whenever 'NAT' is selected, CROCODILE automatically selects 'Connectivity' as a »REQUIRED_MODULE«, too.

4.11 Module 'Passwords'

On an IOS router, various passwords — also called »secrets« with some commands — may be configured, for instance, login passwords, enable passwords or FTP passwords. In order to protect the password settings from disclosure, passwords may optionally be stored in the configuration in encrypted form: IOS offers two encryption variants for this, no. 5 and no. 7. The no. 5 encryption is cryptographically secure, while the no. 7 variant only provides some cover-up for the cursory observer, but can be decrypted without problems.

The 'Passwords' checker module examines the router configuration for insecure passwords or passwords with unnecessarily weak encryption, as well as a missing activation of the password encryption service.

Evaluation criteria

The evaluation criteria in the passwords area are mostly pre-defined in the program code of the checker module; only the candidate words list for the test »search for trivial passwords« may be freely configured in 'Passwords.conf' in the 'Configuration' subdirectory. The most important issues checked by the module are these:

- Are all password codes contained in 'password' or 'secret' clauses valid password strings?
- Are insecure — i.e., easily guessed — passwords used? Are minimum standards for password length and character set observed?
- Are required enable secrets configured?

- Are unencrypted or weakly (no. 7) encrypted passwords used even though strong encryption would be possible?
- Are unencrypted or weakly (no. 7) encrypted passwords re-used in strongly encrypted (no. 5) format, thus compromising the strong encryption?
- Are passwords or secrets used in parallel to other AAA authentication mechanisms (see Chapter 4.8)?
- Is the password encryption service («service password-encryption») of the router activated, in order to hide password settings in the configuration text and in the IOS 'show' commands?
- For which 'enable' levels 0 to 15 are passwords configured?

All findings are collected under the 'Authentication & Authorization' bookmark, subheading 'Passwords and Secrets'.

Password decryption

Passwords encrypted with the no. 7 method may be decrypted with a rather simple procedure. The checker module uses this for the password reuse check. CROCODILE will normally output the decryption result for each no. 7 password as an 'INFO' message in order to point out the problems with such passwords.

In order to maintain the hiding of passwords during analysis, the user may optionally suppress the clear text output of passwords by specifying

```
DISCLOSE_PASSWORDS = NO
```

in 'Passwords.conf'. The corresponding messages will not be included in the result output then. However, the user should be aware that this provides protection against casual spying only; the decryption procedure used by CROCODILE is publicly known and may easily be obtained through the Internet.

The default setting is »DISCLOSE_PASSWORDS = YES«, that is, all decryptions are shown in the evaluation protocol.

Trivial no. 5 passwords

The encryption method no. 5 is secure according to current knowledge; a systematic decryption seems impossible. This is why the 'Passwords' checker module cannot directly check the quality of passwords encrypted by this method. The tests are therefore limited to excluding several trivial passwords.

First, the tool checks whether one of the clear text or no. 7 passwords reappears in no. 5 encryption, too. In this case, the no. 5 encryption would be useless, as the corresponding clear text is easily found. Next, the encrypted no. 5 passwords are compared against a list of trivial clear texts in encrypted form. The list of passwords considered trivial is specified in the 'Passwords.conf' file in the 'FORBIDDEN_PASSWORDS' attribute; the user may extend or change this list at will.

Note: *Encryption method no. 5 is a (purposely) intricate procedure; as a consequence, sequential checking of clear text candidates — known as a »dictionary attack« — requires considerable computation time. Therefore, the user should refrain from extending the 'FORBIDDEN_PASSWORDS' list too much.*

In addition to the password candidates named in 'FORBIDDEN_PASSWORDS', the host name of router, if defined in the router configuration, is checked as a possible password.

To save evaluation time, the user may suppress the dictionary attack on no. 5 passwords by entering the following line in 'Passwords.conf':

```
NO_5_CHECK = NO
```

Depending on the number of 'FORBIDDEN_PASSWORDS' candidates and the frequency of no. 5 encryptions in the router configuration, this may significantly reduce the run time of the checker module. A more scalable option is to simply remove some entries from the 'FORBIDDEN_PASSWORDS' list.

Required no. 5 passwords

In large networks, the configuration of certain enable secrets may be declared mandatory for all routers to ensure that every router is accessible with a known common password.

The user may specify attribute 'REQUIRED_PASSWORDS', a comma-separated list of cleartext passwords, in the 'Passwords.conf' file. If specified, CROCODILE will check that each of the listed password strings will actually appear in no. 5 encoding in the configuration under test. A warning is issued for each missing list member. Note that the tool will not check for required no.7 passwords, simply because no. 7 encoding is considered unsafe and should thus be avoided.

Package 'Digest::MD5'

The encryption method no. 5 is based on MD5, a cryptographic hashing algorithm by RSA Data Security, Inc. To implement the encryption with little

effort, the 'Passwords' checker module makes use of a ready-made module from the Comprehensive Perl Archive Network (CPAN), named Digest::MD5. This package provides the required functionality in a convenient way. The package should be contained in any modern Perl distribution.

If that module is missing from the user's local Perl installation nevertheless, the following warning message appears on the console at program start time:

```
*** Auxiliary::BSDcrypt -- Unable to locate CPAN Module 'Digest::MD5'
*** You may either try to install this module (see www.cpan.org)
*** or use your Perl unchanged: Just ignore this error message!
*** In the latter case, CROCODILE can't perform any No. 5 checks
*** Apart from that, CROCODILE's functionality remains unchanged.
```

In this case, the user may simply continue the program execution; CROCODILE will behave mostly unchanged. Only those checks within the 'Passwords' module that refer to no. 5 encrypted passwords will return no results. All other modules keep their full functionality.

Alternatively, the user may also attempt to install the missing module 'Digest::MD5' from a CPAN server for free. A possible URL for downloading the module is 'www.cpan.org'. This will ensure the full functionality of the 'Passwords' module.

4.12 Module 'CiscoLinks'

The pattern handlers in the various checker modules typically attach references to Cisco's *Online Command Reference* to IOS configuration clauses that trigger the handler. These will appear as »Online Resources« in the lower left frame of the CROCODILE display. To determine and assign appropriate references, the pattern handler calls method 'attach_ciscolinks()' (see Appendix A). However, if none of the specified patterns matches a given IOS configuration clause, then no handler is activated, and therefore no references are assigned. Thus, hyperlinks to Cisco resources are only available if a line is covered by an evaluation rule: Lines that are ignored by CROCODILE (those printed in gray in the main HTML frame) remain unlinked.

The sole purpose of the 'CiscoLinks' module is to assign »Online Resources« even to those clauses that are not covered by other checker modules.

Evaluation criteria

This module does not analyze the IOS configuration, nor does it apply any evaluation rules. Rather, it defines a universal pattern matching arbitrary IOS clauses,

and a corresponding handler whose only task is to call 'attach_ciscolinks()' for each matching line, in fact, for any input line.

By resetting the matching line's 'CRITICALITY' attribute to 'undef', the module signals to the CROCODILE framework that the input line just matched should be regarded as »unseen« although it just triggered the module's pattern handler. Provided that no other pattern applies (or that all pattern handlers that are triggered pretend not to have seen the input line), the IOS clause is treated as »not covered by CROCODILE«. With this technique the module is able to attach hyperlinks without spoiling CROCODILE's coverage indicator.

Remarks

In principle, all checker modules could simply rely on 'CiscoLinks' without bothering to call 'attach_ciscolinks()' on their own. In some cases, however, selecting the most appropriate references requires some context information that can only be provided by deeper analysis. Thus, specialized pattern handlers are generally best suited to guide 'attach_ciscolinks()' in making optimum choices. If different modules suggest different online references, CROCODILE will automatically select the most specific ones and ignore the poorer choices, based on some »best fit« metric.

If there is no need to attach »Online Resources« to unrecognized lines, the user may deselect 'CiscoLinks' in the configuration file 'plugin.conf' to reduce the amount of HTML output. Note that for each hyperlink generated by the module, two (small) HTML files are added to 'HTML_results'. Depending on the number of input lines and the coverage achieved by the checker modules, switching off 'CiscoLinks' may save hundreds of output files, although it offers only marginal performance gains in terms of runtime.

Module 'RATemulation' (see Chapter 5) inspects all input lines, regardless of the specific RAT evaluation ruleset that is configured. For the assignment of online resources, it (mostly) relies on 'CiscoLinks' because it is unclear whether each of the inspected IOS clauses is actually analyzed or not. Therefore, whenever 'RATemulation' is selected, CROCODILE automatically selects 'CiscoLinks' as a »REQUIRED_MODULE«, too.

5 Emulation of the Router Auditing Tool (RAT)

The Router Auditing Tool (RAT) is a tool that checks IOS configurations. It can be obtained from the Center for Internet Security [CIS] as a free download [RAT2.2].

Similar to the CROCODILE module 'CompoundPatterns', RAT reads freely configurable evaluation rule specifications and applies them to the router configuration under examination. The evaluation result consists of a list of all evaluation steps applied and their individual result, either 'PASS' or 'FAIL'. From a weighting of the evaluation rules and the list of results, RAT computes a benchmark score, which is the official CIS router benchmark (see also Figure 23 auf page 96).

RAT is rather well known and it is sponsored by important organizations including the National Security Agency [NSA] and the SANS Institute [SANS]. Due to this sponsoring, elaborate evaluation rule sets are available for RAT, and they are updated over time.

CROCODILE offers a special emulation module to make use of these resources. The module emulates RAT Version 2.2 as of the CIS distribution released in September 2004. Using the RAT evaluation rule sets provided by CIS, our tool is able to compute the CIS benchmark for Cisco IOS router and PIX firewall configurations.

5.1 Emulation module 'RATemulation'

The 'RATemulation' module reads RAT rule specifications and applies them to the evaluation target. 'RATemulation' thus provides freely configurable evaluation criteria within the limits of the RAT format.

In regards to its flexible configuration, the emulation module is similar to the 'CompoundPatterns' module (see Chapter 4.2). In fact the expressive power of the supported rules corresponds rather closely to the capabilities of the »basic patterns« of that module. (Note that CROCODILE's »compound patterns« are even more powerful than this, see Table 2 on page 47.)

Each RAT rule basically consists of a pattern — more specifically: a Perl regular expression — describing an IOS clause. A pattern is marked as either 'required' or as 'forbidden', and an evaluation step fails if either a pattern marked as

required is missing in the router configuration, or a pattern marked as forbidden does appear in the configuration. Thus, these rules are checks for individual, isolated configuration lines, like CROCODILE's basic patterns: Evaluation rules applying to the interplay of several distinct IOS clauses cannot be expressed in the RAT rule format.¹⁶

Similar to 'CompoundPatterns', RAT rules may be restricted to specific configuration contexts (see Chapter 4.2, Section »Contexts« on page 46) such as an interface or line context. RAT offers nothing more than 'CompoundPatterns' in this respect.

In contrast to 'CompoundPatterns', however, RAT rules are equipped with a number of additional attributes, including 'Importance', 'Description' (objective of this evaluation step), 'Reason' (security relevance), 'Fix' (suggested correction in case of a violation of this rule) and 'Warning' (hints about limited applicability of this rule). Such information may be used to further annotate the evaluation report with explanations and tips. This makes RAT rule sets interesting for CROCODILE, too. We have therefore extended our tool by a RAT module.

The ability to process RAT rule sets in the familiar way facilitates the migration to CROCODILE for former RAT users. Users who had specified their own RAT evaluation rules according to their local situation may continue to use their private rule set without change. CROCODILE delivers RAT results in the classical format conforming to RAT as well as in its own characteristic hypertext format — including a comparative juxtaposition of the results of repeated evaluations (see Chapter 7.2, Section »Creating a comparative display of two evaluation results« on page 115). RAT evaluation results are integrated into the CROCODILE evaluation report, and the evaluation result is complemented by RAT-specific hints, suggestions and references.

Note: *In the spirit of other CROCODILE checker modules, 'RATemulation' attaches hyperlinks to »Online Resources« to each IOS configuration clause. For most of these annotations, 'RATemulation' simply relies on module 'CiscoLinks'. Therefore, whenever 'RATemulation' is plugged in, 'CiscoLinks' is automatically plugged-in, too.*

¹⁶ Strictly speaking, there is an option to provide, instead of a pattern, an evaluation routine (to be written in Perl) — a so-called »Callout« — that can implement arbitrary tests on the IOS configuration description. However, RAT offers hardly any support for callout programming, and the evaluation result of a callout is limited to a binary 'PASS' or 'FAIL' again. This option thus seems more like a kludge to circumvent the strict functional limitations of the RAT approach in special cases. The RAT 2.2 standard distribution of September 2004 contains only five rather trivial callouts: CheckBufferingSizeMin(), CheckIOSPasswordQuality(), MatchAtLeast(), CheckIOSSecretReuse(), and CheckExecTimeout(). Only three of these are actually used for the CIS benchmark.

5.2 RAT conformity

When developing the emulation module, we strived to perform all evaluation steps in a form equivalent to that of RAT. However, due to copyright and licensing restrictions, only tools or auditors accredited by CIS are allowed to officially assign a CIS benchmark score, and CROCODILE is not accredited at this time. Users may thus decide on their own how to judge the conformity of the CROCODILE emulator and how to estimate the value of its results. We claim that the 'RATemulation' module faithfully applies RAT evaluation rule sets.

However, it should be noted that the rule processing under CROCODILE does actually diverge from that of RAT in some respects. Minor differences are mostly owed to several weaknesses of the RAT software. When analyzing the RAT distribution 2.0 of March 2003, we found, for instance

- *typos* in the rule specifications that are silently ignored by RAT but may effect the rule application
- *unexpected matches* of certain rule patterns where RAT applied unwarranted rules in test examples
- *missed matches* of rule patterns where RAT failed to trigger the appropriate rule in test examples
- *inconsistencies* like cross references to undefined rule attributes that RAT did not flag nor apparently even notice

We have reported these shortcomings to the RAT development team. Unfortunately, a closer inspection of RAT 2.2 revealed that some deficiencies still exist in the current version. At any rate, the emulation module has been designed so that it avoids the known errors of RAT and correctly applies patterns, sometimes even automatically correcting incorrect patterns — such as Perl regular expressions that are apparently too greedy (cf. Section »Conflicts: Greedy or minimal pattern interpretation?« on page 21); CROCODILE lists recognized errors and inconsistencies in RAT rule data at the console. If a continued working seems meaningful after discovering such a problem, CROCODILE continues the RAT evaluation step in spite of the warning message and often arrives at a meaningful result nevertheless.

Since we had to diverge slightly from the behavior of the available RAT distribution in order to avoid errors, small differences may result between RAT and CROCODILE evaluation results. However, the user may easily verify that this effect does not usually affect the evaluation result in any considerable way.

5.3 Configuring the emulation module

The emulation module receives its detailed evaluation instructions via the configuration file 'RATemulation.conf' in the 'Configuration' subdirectory (see the directory structure in Figure 10, page 39). This file determines

- the path names of the RAT rule specifications and corresponding rule meta data
- the location of any callout code (cf. Footnote 16 on page 90) for RAT rules that are based on rule-specific Perl code.

Figure 22 shows a typical module configuration. It may be assumed that users interested in RAT have already downloaded and installed a copy of the RAT distribution. If RAT is installed, the paths to be configured result naturally from the structure of the RAT root directory. However, the user may also store the required data in other places, such as the CROCODILE configuration subdirectory.

Note: *For copyright reasons, the CROCODILE distribution does not contain RAT context, field or rule descriptions, and no callouts. All these are included in the RAT distribution, which users may obtain directly from the Center for Internet Security [CIS]¹⁷ or create on their own in order to operate CROCODILE's 'RATemulation' module. If CROCODILE does not find the required data among the paths configured, it will report an error and disable the RAT emulation automatically. The evaluation then continues with the remaining checker modules in the usual manner.*

The following parameters must be set (for more information on the attributes mentioned below the reader is referred to the original RAT documentation):

- **RATLIB:** <path to lib directory of a RATv2.2 distribution>
If callout rules are employed, this indication is required as some callouts use data declarations of the RAT standard distribution; if this setting is missing or access to the RAT library fails, callout rules will be skipped with a warning message in the later evaluation run. Conventional RAT evaluation rules should work nevertheless.
- **CONTEXTS:** <path to file with IOS context descriptions>
The RAT standard distribution stores the context specifications for IOS in the 'etc/configs/cisco-ios' subdirectory in the 'contexts.txt' file. The specifica-

¹⁷ Note that CROCODILE will report a few ruleset integrity violations because the original RAT distribution contains a few — rather insignificant — defects.

```

# Where is the RAT library (required for callout execution)?

RATLIB = /rat2.2/lib

# Where is the specification of IOS contexts?

CONTEXTS = /rat2.2/etc/configs/cisco-ios/contexts.txt

# Where is the specification of valid rule formats?

FIELDS = /rat2.2/etc/configs/cisco-ios/fields.txt

# Where are the specifications for RAT Globals, RAT Parameters,
# and RAT Evaluation Rules?

RULES = /rat2.2/etc/configs/cisco-ios/common.conf,
        /rat2.2/etc/configs/cisco-ios/cis-level-1.conf,
        /rat2.2/etc/configs/cisco-ios/cis-level-2.conf,
        /usr/local/rat/my-wonderful_private_ruleset.conf

# Where is the NSA Router Security Configuration Guide
# (to be hyperlinked to RAT rules)

SECURITY_GUIDE = /rat2.2/rscg.pdf

# Do you prefer the original RAT scoring method (which is rather
# obscure)?

WEIRD_RAT_SCORING = YES

# Should we add page labels to PDF hrefs? This is convenient but
# does not work for all browsers.

PDF_PAGE_LABELS = NO

```

Figure 22

Sample settings for 'RATemulation.conf'

tions determine what analysis contexts RAT can distinguish and how they are named.

- **FIELDS:** *<path to file with field format descriptions>*
The RAT standard distribution stores its field format specifications in the 'etc/configs/cisco-ios' subdirectory in the 'fields.txt' file. These descriptions determine the attributes that are required or allowed in RAT rules as well as the domains of each attribute.
- **RULES:** *<List of paths to rule files>*
The RAT standard distribution contains some standard rule sets in the

'etc/configs/cisco-ios' subdirectory in the 'common.conf' file (basic declarations), 'cis-level-1.conf' (evaluation rules for fundamental security requirements) as well as 'cis-level-2.conf' (evaluation rules for advanced security requirements). RAT expects user-specific additional rules in the file 'local.conf' in the same directory. An arbitrary number of paths to arbitrary directories may be listed under 'RULES'.

- **SECURITY_GUIDE:** *<path to Router Security Configuration Guide>*
The RAT standard distribution contains the Router Security Configuration Guide [NSAGuide] under 'rscg.pdf'. If an invalid path is given here, CROCODILE cannot generate the corresponding hyperlinks as cross references as is required by the standard rule specification. However, this does not affect the evaluation result itself.
- **PDF_PAGE_LABELS:** 'yes' or 'no'
If 'yes' is given here, CROCODILE will generate HREFs to PDF files referring to a specific page within the PDF documents. The format of a reference is '[- **WEIRD_RAT_SCORING:** 'yes' or 'no'
If 'yes' is given here, CROCODILE will compute the score of the CIS benchmark in the same way as the RAT software for compatibility reasons. Independent of that, CROCODILE will compute another benchmark score in a modified way, as the original formula seems questionable \(see Figure 23 on page 96\): More on this in the following section.](<file name>.pdf#page=<page no>)

5.4 Benchmark computation (CIS benchmark and CROCODILE score)

The RAT framework in its entirety realizes the security benchmark for IOS router configurations as suggested by CIS. For easy comparison, all evaluation findings are condensed into a single benchmark score in the interval of [0 ... 10]. The closer this result approaches the ideal value of 10.0, the more secure the router configuration is considered. Unfortunately, the benchmark is computed by the following, rather questionable algorithm:

1. Each evaluation rule is applied to all IOS clauses matching the rule's pattern. Each such rule application is counted as a so-called »check«.
2. Each »check« with a 'FAIL' result is counted under the »checks failed« category.

3. If an evaluation rule is applied several times and some »checks« deliver 'FAIL', while others deliver 'PASS', the successful rule applications will be ignored. Only those »checks« with a 'FAIL' result will be counted under the »checks failed« category.
4. If an evaluation rule is applied one or more times and all »checks« deliver a 'PASS' result, this is nevertheless counted as only one successful »check« under the »checks passed« category. That is, repeated success of the same rule will be ignored in any case.
5. The ratio of the sum of all weights (rule attribute 'Importance') of all »checks passed« to the sum of all weights of the »checks failed« and »checks passed« constitutes the desired benchmark value.

Steps 3 and 4 of the algorithm obviously put emphasis on failed in contrast to successful rule application. This may result in a rather distorted picture: If, for example, 9 of 10 interfaces have been configured correctly and only one incorrectly with respect to a rule, the RAT evaluation report will record exactly one failed rule application — the larger part of the successful rule applications is not counted at all, and not even recorded in the evaluation report! If this were the only rule activated in this evaluation run, a RAT score of 0.0 would result, even though the configuration is next to correct.

To avoid such distortion effects, CROCODILE will report, in addition to the original benchmark score according to CIS specifications, another benchmark score resulting from an unbiased counting of *all* failed and successful »checks«. In contrast to RAT, CROCODILE records all »checks«, each 'PASS' and each 'FAIL'. The CROCODILE evaluation report is thus more complete, and the CROCODILE score will usually be slightly higher than its CIS counterpart.

5.5 Evaluation reports

The emulation module generates the evaluation reports in a format conforming to that of RAT (with some additional information added), such that a user should master the migration from RAT to CROCODILE without problems. Figure 23 shows the typical format of such a CROCODILE-generated RAT report. Lines referring to specific rules are hyperlinked to the corresponding rule description including any appropriate cross references.

A summary of the RAT-specific results is available under the »RAT Emulation« bookmark in the hypertext report. Additionally, CROCODILE assigns every RAT finding to the various views and categories of CROCODILE's native result format. Thus, RAT findings are treated like ordinary CROCODILE findings. For example, if a line, a clause context, or a logical context is affected by a RAT finding, that finding will be integrated into the corresponding line, clause or context displays,

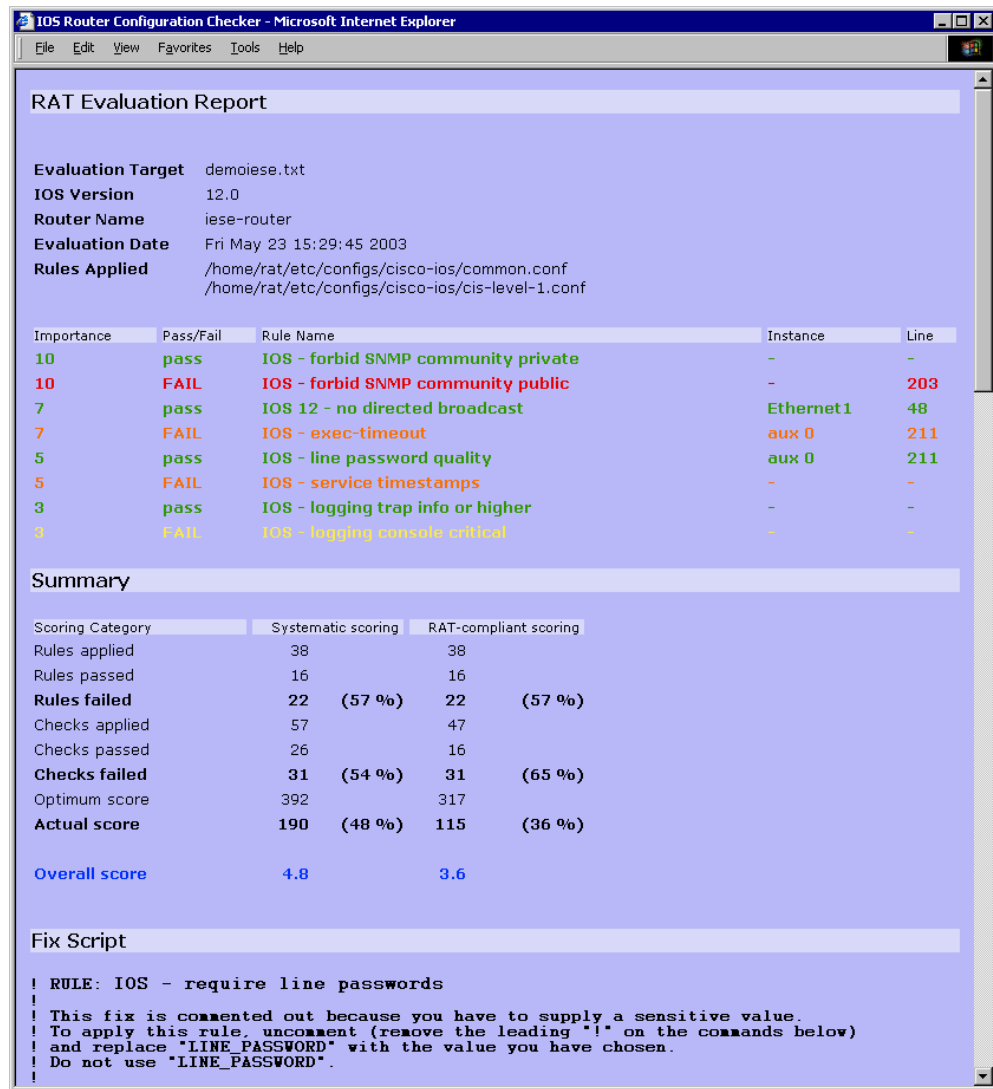


Figure 23 Excerpt from a typical RAT evaluation report as generated by CROCODILE

including the »Evaluation Report« overview display. Further, a profile bar for the RAT module is added to the »Evaluation Profile«.

The duplicated reporting in native RAT as well as native CROCODILE format delivers the familiar evaluation report to all users, including former RAT users.

5.6 Rule descriptions

RAT evaluation rules may be ordered hierarchically in classes and subclasses. A RAT rule class corresponds roughly to an »evaluation <XXX> topic« in CROCODILE parlance. The tree structure of the rules hierarchy supports selecting or deactivating whole subtrees. For this purpose, each rule class and each rule description has an attribute 'Selected' with possible values 'YES' or 'NO'.

Original RAT contains its own editing tools for rule sets supporting a convenient selection and activation of individual evaluation topics. The rule editor ensure that the corresponding rule attributes are set correctly and consistently.

In addition, many evaluation rules may be parameterized by variables. This may be used, for example, to adapt rules to locally assigned network addresses. The corresponding variable bindings are part of the rule specification and must be adapted by the user before running RAT with such a rule. The RAT rule editor will support the user in parameter assignment, too.

CROCODILE recognizes rule classes and rule hierarchies as well as configurable rule parameters. It will check their consistency and applies the chosen settings correctly. For example, if a rule class is deactivated using »Selected: no«, all rules belonging to this class will be automatically ignored, even if individual rules of the class are (inconsistently) activated by »Selected: yes«.

Under the »RAT Emulation« bookmark the »Rule Descriptions« link points to a comprehensive description of all configured rule specifications. The description is given as a hypertext document and contains all rule classes, rules, and parameters with their relevant attributes. The rule hierarchy is represented by hyperlinks that are assigned to class names and rule names. This enables easy navigation within the tree of evaluation rules. Deactivated parts of the rule specification are shaded.

However, CROCODILE does not provided any editor support for RAT rule sets: Users will normally have at hand the RAT editor from the original distribution. If necessary, rules may also be edited quite simply — although less conveniently — with a conventional text editor.

For further details about the exact rule description format and the available settings, the reader is referred to the documentation of the original distribution [RAT2.2].

5.7 Concluding remarks

The RAT emulation mode enriches CROCODILE in several respects:

- It offers an easy migration to CROCODILE for RAT users, while preserving the investments in the definition of custom RAT rule sets.
- It extends the CROCODILE hypertext report by further cross references and fix suggestions.
- It ensures participation in evaluation rules, callouts, fix suggestions and literature references that CIS or others may publish for RAT in the future.

Conversely, CROCODILE may be an enrichment for RAT, as it avoids or compensates for some of the shortcomings of the current RAT distribution, and as it delivers more extensive evaluation reports with the same set of rules.

From a CROCODILE perspective, specification options at various levels are now available to the user:

- Simple and obvious evaluation criteria requiring no further explanation may be expressed in the fastest and most convenient way using the 'CompoundPatterns' module. Using compounds, even rules spanning several configuration clauses may be specified, supporting even conditional rules, variable binding, and quantors.
- Evaluation criteria that are simple to check but not obvious to grasp and therefore require further explanation are best expressed in RAT format. The advantage of this rule format is that cross references, explanations, and fix suggestions may be added without programming effort — the downside is the relatively high configuration effort and the substantially weaker expressiveness relative to 'CompoundPatterns'.
- The specification format for 'IngressEgress' is specifically tailored to ACL evaluation criteria and enables options for evaluation rules far beyond the capabilities of either 'CompoundPatterns' or 'RATemulation'.
- Regarding evaluation criteria that cannot be expressed with any of the available configuration formats, the CROCODILE framework provides a rather powerful programming interface to implement new checker modules. Such modules may be configurable in a format tailored to the specific evaluation topic.

Programming new checker modules will remain the last resort for the average user: In many cases the available configuration formats will suffice to express the desired evaluation criteria.

We expect further configurable checker modules for CROCODILE to appear, and more flexibility for existing configuration options in the future. At present,

'IngressEgress', 'CompoundPatterns' and 'RATemulation' offer interesting possibilities to find a balance between universality, expressiveness, and convenience of a specification format.

6 Batch Processing

CROCODILE is primarily an interactive tool to perform a detailed evaluation of a single IOS configuration. However, operators of large networks who monitor numerous IOS routers occasionally need to perform large-scale evaluation campaigns. This requires batch processing in unattended OFFLINE mode.

CROCODILE supports operation in batch mode by means of a separate front-end program: 'batchcroco'. In this chapter we describe the batch processing mode and explain the available processing options.

6.1 Using CROCODILE in batch processing mode

In batch mode CROCODILE analyzes several IOS configuration files in a row, generating one evaluation report per input file and summarizing an aggregate evaluation result in tabular form.

The 'batchcroco' program in the CROCODILE's main directory performs these functions. The program is invoked with a list of input file names as its invocation parameters. This list of evaluation targets may also contain directory names: In this case, all files in the named directories are subject to batch processing. Further subdirectories may be included recursively in the processing on request.

Note: *Depending on the operating system, file names may also contain spaces. In order to unambiguously interpret file name lists as invocation parameters, such file names — or at least their whitespace characters — must be suitably quoted or »escaped« according to the conventions of the operating system, such as by single or double quotes or by a preceding backslash.*

Patterns containing wildcard characters such as '*' and '?' may also be used. Typical Unix shells resolve these and other (even more complex) search patterns immediately and pass a list of completely resolved file names to the application program. Windows command prompt windows, however, pass the unresolved wildcards to the application. If CROCODILE discovers wildcards in file names that were not resolved at invocation time, it will interpret them as search patterns and resolve them by itself.

Note: Depending on the operating system, file names are case-sensitive or case-insensitive, sometimes even irregularly depending on the application. The resolution of search patterns thus sometimes produces all »hits« independent of character case, and sometimes only those with matching case. Users should therefore make sure that wildcards are resolved on their platform according to their expectations.

For instance, in order to analyze all files ending in '.txt' in the 'SampleConfigs' directory, the correct invocation on a Unix operating system is

```
user@host[CROCODILE] perl batchcroco SampleConfigs/*.txt
```

```
CROCODILE batch processing started.
```

```
=====
=
= Processing 20 batch jobs          2003/09/25   14:16:08
=
=====
--- Job 1 : SampleConfigs/acsac.txt -----
--- Job 2 : SampleConfigs/brian1.txt -----
--- Job 3 : SampleConfigs/brian2.txt -----
...
--- Job 19: SampleConfigs/sft.txt -----
--- Job 20: SampleConfigs/some1.txt -----

Writing summary to '/tmp/batchresults/summary.html' ...
FILENAME      FINDINGS    OKAYS  INFOS  CHCKS  WARNS  ALRTS  PROBMS  ERRS
ibmt.txt      165         40    31     31     51     12
acsac.txt     168         65    30     27     30     16
ict.txt       167         49    28     29     50     11
ciosc.txt     168         70    29     26     30     13
...
ise.pix.txt   161         23    69     18     41     10      6
some1.txt     156         54    24     29     39     10
brian1.txt    189         70    36     33     41      9
brian2.txt    185         63    42     30     41      9
sft.txt       103         16    22     27     34      4
```

```
Writing all findings to '/tmp/batchresults/allfindings.txt' ...
```

```
Batch process terminated.
```

```
Batch jobs with PROBLEMS: 1  ERRORS: 0  ABORTS: 0  INTERRUPTS: 0
Overall user time 13:35 min  system time 0:12 min
```

The program creates the result files in a directory configurable by the user; in the example above, this is '/tmp/batchresults'. The result of batch processing includes

- a protocol of the complete batch run: 'batchprocess.log';

- a table with the individual results of all batch jobs: 'summary.<format>';
- one evaluation report per input file: '<file name>.<format>';
- optionally, a list of all findings of all batch jobs in machine-readable text format: 'allfindings.txt'.

The file name extension of the summary table and the individual reports depends on the desired output format — either ASCII text ('txt') or HTML ('html'). The optional file 'allfindings.txt' is created in text format only; HTML output is not supported.

Figure 24 shows two typical examples of such output files. The entries in 'summary.html' (upper right) are hyperlinks. By clicking on a table row, the corresponding report with the detailed findings will open (displayed in the lower left in Figure 24). The HTML evaluation report generated in batch mode is identical to the 'Evaluation Report' created by program 'crocodile' in interactive mode. However, 'crocodile' will create further views and additional information that are not generated in batch mode using 'batchcroco'.

Evaluation results in ASCII format are created in batch mode only. In interactive mode, only HTML and XML format is available.

Failure or abort of individual batch jobs

The 'batchcroco' program is only a front end that ultimately passes the individual jobs to 'crocodile' with suitable invocation parameters. An individual job may produce warnings or error messages. Such »exceptional events« are noted in the summary display and recorded in the log file 'batchprocess.log'. Figure 24 shows an example: Six »problems« have occurred during the 'iese.pix.txt' batch job. The final message from the batch processing indicates the number of problems and errors that occurred.

An error may be so grave as to make a meaningful completion of the job impossible. In this case, the concerned job is aborted in a controlled way, and is marked as »aborted« in the summary table. No findings are reported for this job. However, batch processing continues normally with the next job. CROCODILE indicates the number of aborted jobs in a final message on the console.

User-controlled job abortion

The current batch job may also be cancelled interactively using the 'Control-C' key combination. The job will then be terminated without any findings and marked as »interrupted« in the summary. The application will then prompt the

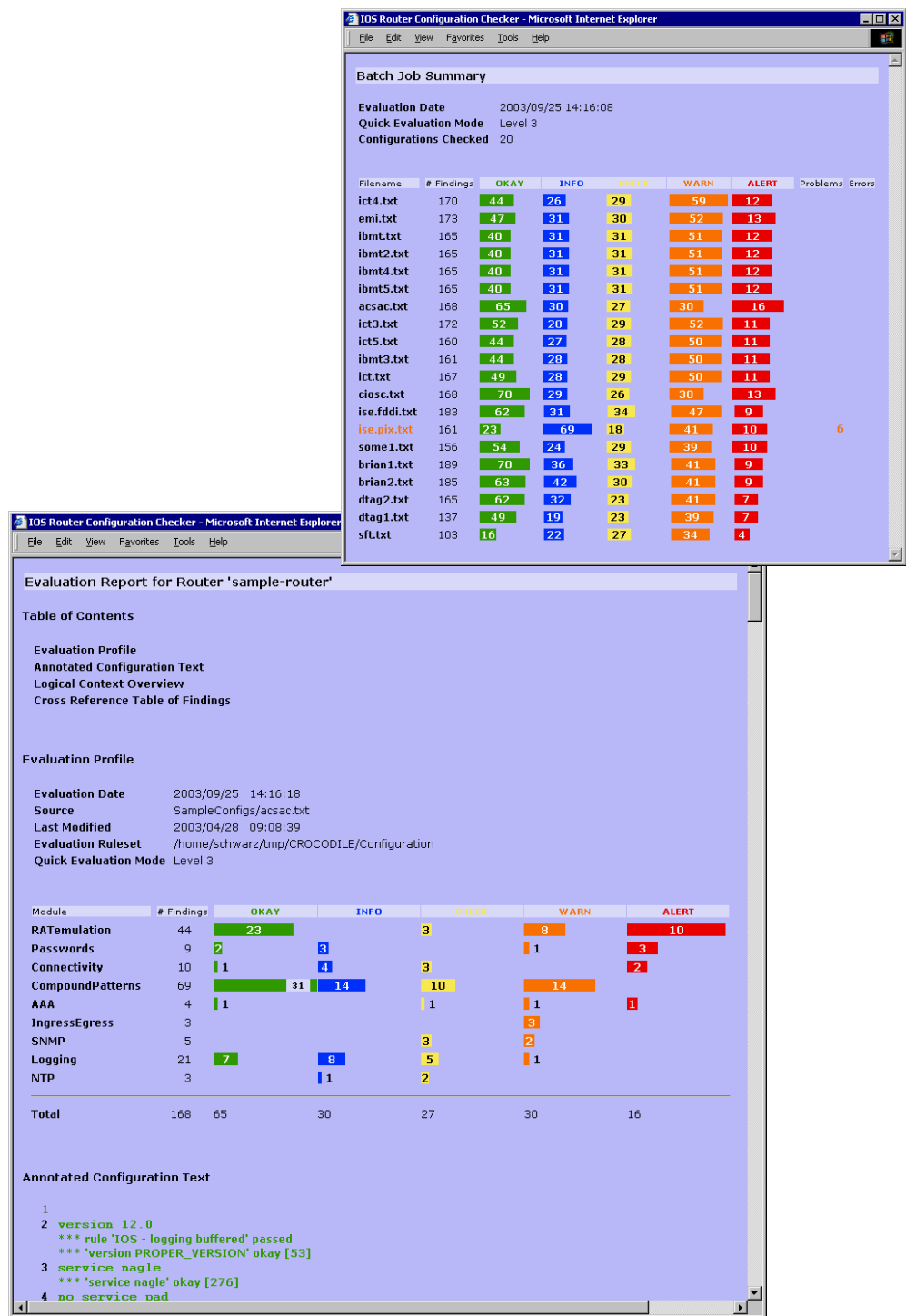


Figure 24 HTML results of the batch processing run: 'summary.html' and 'acsac.html'

user whether to continue with the next job. Unless the user explicitly denies this question by typing 'n', the batch process will be continued with the next regular job.

Cancelling a job using 'Control-C' allows to skip individual jobs without discarding the results of earlier jobs. The number of aborted jobs is indicated in the final console message.

Note: *This feature does not work under all operating systems. If CROCODILE is started from a command shell under Microsoft Windows, Windows will not properly pass on the interruption signals to the Perl run-time system in the case of multiple job cancellations. Catching and processing the second and all further 'Control C' signals is thus not under the control of CROCODILE under Microsoft Windows.*

6.2 Configurable options

The front end 'batchcroco' may be used without any further invocation options. The program assumes default settings that usually produce reasonable results. These settings may optionally be changed by entering the desired parameter values in the 'batch.conf' file in the 'Configuration' subdirectory. Parameters not explicitly set in that configuration file will get their default values. Table 3 lists all configurable options and their defaults.

All settings accessible via 'batch.conf' may also be set interactively using command line options for a single invocation. A command line option will override the corresponding option in 'batch.conf'. Table 3 shows the correspondence of configuration parameters and command line options.

As an additional command line option, '-h', that causes 'batchcroco' to print an online help message. Not surprisingly, this option cannot be configured.

6.3 Choosing evaluation rule sets

Depending on its deployment context, a router must be configured differently. Accordingly, the evaluation criteria will also differ from case to case: What is good and important for a backbone router may be a clear violation of security policies for an access router — and vice versa! This requires changing evaluation criteria for each batch job.

CROCODILE manages its configurable evaluation rule sets in subdirectories. In each such subdirectory, the various checker modules have their own configuration files. That is, each checker module manages its evaluation criteria individually, independent of all other modules. A checker module '<module>' typically has a

option in 'batch.conf'	command line	meaning	default
ResultDir	-d	directory where all result files are created	Current Working Directory
OutputFormat	-f	output file format: <i>HTML</i> or <i>ASCII</i> , or both	<i>ASCII</i>
FieldSeparator	-fs	separator for fields in ASCII output files: <i>s</i> (= <i>SEMICOLON</i>), <i>t</i> (= <i>TABULATOR</i>) or "<string>"	<i>TABULATOR</i>
IncludeSubdirs	-i	Recursively process all subdirectories of each input directory: <i>on</i> or <i>off</i>	<i>off</i>
ResultOverwrite	-o	Overwrite existing result files without warning: <i>on</i> or <i>off</i>	<i>off</i>
ResultPrefix	-p	Prepend the indicated string as <Prefix>- to all output files	– no prefix –
Quick	-q	Skip some non-essential checks in batch mode: <i>0</i> (= <i>off</i>), <i>1</i> , <i>2</i> or <i>3</i> (= <i>on</i>)	<i>on</i>
QuoteFields	-qf	Enclose all text fields with double quotes in the ASCII output: <i>on</i> or <i>off</i>	<i>off</i>
Ruleset	-r	Subdirectory name of the evaluation rule set (in the RulesetDir upper directory) to be applied to all batch jobs	derived from the name prefix of the input file "<ruleset>-<filename>", or 'Configuration' by default
RulesetDir	-s	Name of directory containing the rule sets (as subdirectories)	Current Working Directory
ResultAddDate	-t	Add date in -YYYYMMDD format to the names of all output files: <i>on</i> or <i>off</i>	<i>on</i>
Verbose	-v	Level of detail of console output: <i>0</i> (= <i>off</i>), <i>1</i> or <i>2</i> (= <i>on</i>)	<i>off</i>

Table 3 Available options in 'batch.conf' and their corresponding command line options

configuration file '<module>.conf' in the active evaluation rule set directory. Further, the local 'plugin.conf' file determines, which checker modules will be activated during an evaluation run at all. Evaluation rules are thus managed using the schema illustrated in Figure 25. This facilitates maintenance of the software and the evaluation rule sets.

In interactive mode, CROCODILE uses the default evaluation rule set in subdirectory 'Configuration' in the '<CROCOHOME>' directory, the installation directory of

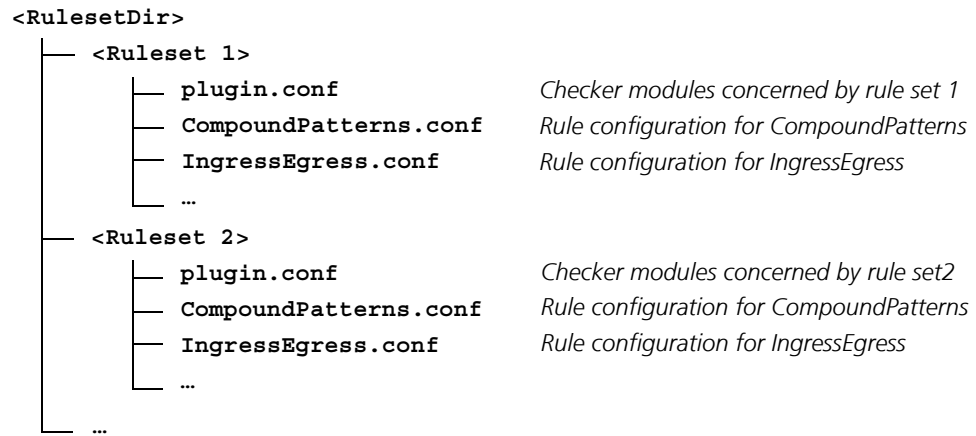


Figure 25 Structuring evaluation rule sets in a directory tree

the tool. The user may change this default setting by configuring a different evaluation rule directory and a different evaluation rule set subdirectory therein. This is controlled by the '-c' option of the interactive front end 'crocodile'.

In batch processing mode using 'batchcroco', an arbitrary evaluation rule set may be set in a similar way, using the `<RulesetDir>` and `<Ruleset>` options in Table 3. The configuration directory 'Configuration' may serve as a template for such a configuration directory. If the evaluation rule set is set using these options, it has effect for all batch jobs in the same way.

If, however, each job is to use an individual evaluation rule set, evaluation rules may be chosen via the name prefix of the input files. Whenever an IOS configuration file named '`<prefix>- ...`' is to be processed, 'batchcroco' will extract the prefix preceding the hyphen from the file name and select the `<prefix>` subdirectory in the `<RulesetDir>` directory as the active `<Ruleset>` for this job. For instance, the invocation

```
user@host[CROCODILE] perl batchcroco MyRules-acsc.txt -s /Rules
```

will analyze the file 'MyRules-acsc.txt' according to the rule set '/Rules/MyRules' — unless, of course, the `<Ruleset>` configuration attribute (or '-r', respectively) are specified. If a rule set was explicitly specified, a prefix in file names will always be ignored.

Note: If files are processed in batch mode whose names incidentally contain a hyphen '-', the prefix before the hyphen will be interpreted as a rule set indicator, unless the `<Ruleset>` configuration attribute is set. This may not always be the intention of the user. If a rule set with such a wrong »rule set name« is found

under <RulesetDir>, the job will be aborted, and the following error message is printed:

```
--- Job 1 of 1: SampleConfigs/b-gw12.txt -----
*** ERROR: Can't find ruleset 'b'.
*** ERROR: File 'SampleConfigs/b-gw12.txt' skipped.
```

The path to the <RulesetDir> rule set directory may be given absolutely or relatively to the working directory of the user. In addition, the path may be given relatively to the CROCODILE installation directory. The keyword '<CROCOHOME>' at the beginning of the path name will have this effect.

NOTE: *The batch processing frontend first looks for configuration files '*.conf' in the configuration subdirectory as defined by the command line options 'batchcroco -s <ruleset directory> -r <ruleset>' and their counterpart 'crocodile -c <configuration directory>'. If a specific configuration file is not provided there, CROCODILE will next search for the file in the default configuration subdirectory '<CROCOHOME>/Configuration'. By placing a plugin configuration into the default subdirectory, the user may share it across multiple rule sets, while selectively overriding other configuration files in each individual rule set.*

Using specific evaluation rulesets for different routing equipment

Ruleset directories (option '-s') and rulesets (option '-r') are particularly useful to support individual evaluation criteria for different types of routers. For example, we used rulesets to add specialized plugin modules and evaluation rules for Redback's SmartEdge family of routers to the CROCODILE framework.

To differentiate between IOS and SmartEdge OS analyses, we simply put the configuration files for SmartEdge-related plugins in a separate subdirectory (e.g.: '<CROCOHOME>/SmartEdge/Configuration'). To evaluate SmartEdge OS evaluation targets, we simply call CROCODILE as

```
user@host[/tmp/CROCO] ./crocodile -c ./SmartEdge/BasicRules ...
```

interactively, for instance, or

```
user@host[/home/jones] /tmp/CROCO/batchcroco -s /tmp/CROCO/SmartEdge
-r BasicRules ...
```

in batch processing mode. The SmartEdge ruleset contains its own 'plugin.conf' and 'batch.conf' configuration files for the selection of SmartEdge-specific plugin modules. Furthermore, it comprises tailored module configurations for each plugin module, for example, a configuration file 'CompoundPatterns.conf' that contains pattern definitions in SmartEdge OS syntax.

6.4 Machine-readable result formats

Using the '-f ASCII' option (if needed, in addition to the '-f HTML' option) the user may tell 'batchcroco' to generate its evaluation reports in ordinary text format. This output format is less readable for a human than the corresponding HTML output, but is more suitable for further processing. Unix systems in particular come ready with numerous filtering tools for quick and easy processing of text data. Most word processing, spreadsheet and mathematical programs support a text import interface, too.

CROCODILE optionally creates its text output in »comma-separated vector« format (CSV), that is as lines of the form

```
<text field 1>,<text field 2>,<text field 3>, ...
```

The separator between the text fields does not have to be a comma, though. Using the '-fs' (field separator) option, any separator character — if needed, even a separator phrase — may be configured. 'batchcroco' uses a tabulator character as the default separator. This format is more or less readable for humans, too; further, it may be directly imported by Microsoft Excel, resulting in a proper table format there. A different separator may be used for import by other tools; comma and semicolon are the most common characters. In particular, a different separator character is necessary if tabulator characters appear within the text fields.

Some tools require that the text fields are enclosed in quotes, in addition to the separation. This will usually prevent problems caused by special characters within the text fields as well. CROCODILE will produce such quotes if the '-qf' (Quote-Fields) option is used.

Using '-f', '-fs' and '-qf', the user may flexibly generate variants of the CSV format. Further processing of the results (e.g., in the form of business graphics, filtering, aggregation, sorting) stored in the files

- summary.txt,
- allfindings.txt

as well as in the individual ASCII evaluation reports may thus be added with minimal effort.

6.5 Logging the batch processing

The status of the batch processing run is indicated at the console. The default configuration of CROCODILE in batch mode is rather silent: The console output is

limited to reporting important problems and indicating the progress of the batch processing.

If problems appear during batch processing, the user may optionally enable a more verbose console output, in order to pin down the origin of the problem. The 'Verbose' configuration attribute ('-v') controls the verbosity of output. Three levels of verbosity are available:

- level 0 (-v off) — minimal verbosity
At this level, only essential messages of the 'error' and 'warning' type as well as the progress of the computation is output.
- level1 (-v 1)
This level adds system messages of the 'info' type to the console output.
- level2 (-v on) — maximum verbosity
At this level, all system messages are output to the console: 'error', 'warning', 'info' and 'verbose'.

The desired level may be specified as number 0 ... 2, or as 'on' or 'off', respectively.

Independent of the 'Verbose' parameter setting, all system messages are logged in the 'batchprocess.log' file. This file resides in the configured output directory <ResDir>. Even with the least verbose console output, the log file contains a complete execution trace according to level 2. The name of the log file may vary depending on the configuration options <ResAddDate> ('-t') and <ResPrefix> ('-p') by a prefix and an appended date (see Table 3). The log data is thus unambiguously linked to a certain evaluation run.

7 Utilities

Subdirectory 'Utils' contains some useful tools supporting and complementing the application of CROCODILE's main application. These utilities are not required for an evaluation run. They were separated out as distinct applications in order not to overload standard evaluation runs.

7.1 Analyzing filtering properties of ACLs and interfaces using 'blackwhite'

Sometimes it is useful to determine exactly what kinds of IP traffic are passed through or rejected by a given access control list. We refer to the set of all IP packets passed through as the »whiteset« of that ACL, and to the set of rejected ones as the »blackset«, correspondingly. Another useful concept is the set of all IP packets that may be propagated through a particular interface of the router, which is called the »propagate set« of an interface.

CROCODILE is capable of generating a complete listing of all whiteset, blackset and propagate set packet formats. Depending on the size and complexity of the ACL, however, this may require a substantial computation effort; moreover, the sets may turn out so large and scattered in certain cases that they deliver only small gains in insight to a human user, no longer worth the immense effort required for their interpretation¹⁸. This suggested separating this part of the analysis as a stand-alone tool — the 'blackwhite' utility.

Application of 'blackwhite'

Using 'blackwhite' requires a previous 'crocodile' evaluation run with the 'Connectivity' checker module. If 'Connectivity' is deactivated, the necessary raw data for an ensuing deeper analysis by 'blackwhite' will be missing. The user will thus perform a standard analysis first with 'Connectivity' being activated. This will store information about the configured ACLs and interfaces in the 'XML_data' subdirectory as XML files.

¹⁸ There is in fact a correspondence here: The longer it takes to compute a set, the more likely the user will not be able to make much use of the result.

Once 'crocodile' has been terminated, the user may start the script 'Utils/black-white.pl' which has no invocation parameters. The utility will find the data stored in the 'XML_data' subdirectory and offer the user a menu of the available ACLs and (line) interfaces. The user interactively chooses an ACL, an interface or a line interface and associated processing options. The utility will then compute the whiteset and blackset of the ACL, or the propagate set of the interface, respectively. The result is printed to STDOUT. Using shell operators, the user may redirect this output to a text file if required.

The choice of restrictions

Whitesets, blacksets and propagate sets are usually determined relative to the set of all IP packets, that is »ip any any«: What protocols, origin addresses, origin ports, destinations, and destination ports will be passed through? An exact answer often cannot be expressed in a few clauses. In the worst case, the resulting sets may only be characterized by a long list of individual special cases.

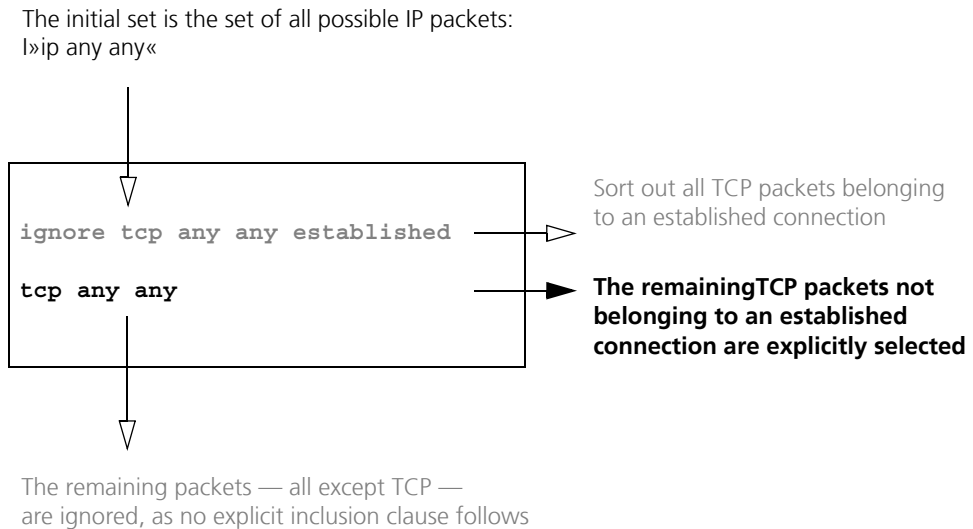
If the result turns out to be too long, it will be of limited use only, and the required computation will often take rather long.¹⁹ Under such circumstances, it is often useful to determine the filtering effect of ACLs and interfaces relative to a restricted base set, for example

- Which TCP connections that are not already »established« may pass?
- In what way is the access restricted from host 6.7.8.9 to privileged destination ports in the '1.2.3.*' and '5.*.*.*' subnets?
- What kind of ICMP 'echo' packets are allowed to pass?

Using restrictions of this kind, the user may narrow the analysis to arbitrarily tailored base sets. This will normally reduce the required computation time, and the structure of the result set will be more comprehensible. If an analysis based on »ip any any« turns out to be unsatisfactory, the user may simply abort the computation and initiate more focused analyses of relevant subsets instead.

Suitable restrictions are specified more or less in the same way as with the 'IngressEgress' module; refer to section »Principle of filter specification« (page 59 ff) as well as Figure 13 to 15. The user characterizes the desired base set by a sequence of ACL clauses that progressively select or discard desired subsets from the

¹⁹ Depending on the performance of the hardware and the problem complexity, computation times of several seconds to minutes may be expected. Even the most difficult cases, however, should be done in less than 10 minutes on a machine with gigahertz clock rates.

**Figure 26**

Sample restriction: »all TCP packets without the 'established' attribute«

universal base set of »ip any any«, leaving the resulting rest open to further selections and rejections.

Example

A simple example for a restriction that may be realized with two clauses is shown in Figure 26. The two restrictions together characterize the base set as the »set of all IP packets not part of an established connection«. Obviously, the restriction clauses have roughly the same format as the corresponding IOS clauses for ACL configuration. However, the 'blackwhite' utility omits the 'permit' keyword, and instead of a 'deny', the keyword 'ignore' is used ('deny' may be used synonymously), which makes the specification even more intuitive. The protocol of a 'blackwhite' analysis is shown in Figure 27.

7.2 Reformatting the evaluation report using 'make_html'

All analysis results are collected in the central database of the tool, which is stored in XML format in the file 'results.xml' at the end of the evaluation run. Additionally, the HTML display component is activated. This creates a hypertext evaluation report (see Figure 6, page 33). Using the utility 'make_html', the user may reformat the XML result file of a finished evaluation run ('results.xml') into HTML hypertext with changing display parameters without having to repeat the evaluation.

```

Select an access control list:

[1] ACL 1
[2] ACL 102
[3] ACL 105
[4] ACL 106

Which one ?: 2

Reading ACL rules '../AclRules/acl_102.xml': ... 5 rules read.

You may restrict the context for which Blacklist/Whitelist analysis.
Restrictions accumulate and may have one of these forms:

[ignore] {any|IPADDR [WILDCAR_DMASK]}
[ignore] PROTO SUBNET [PORTSPEC] SUBNET [PORTSPEC|ICMPSPEC] [established]

Any context restrictions (RETURN to quit)?
?: ignore tcp any any established
?: tcp any any

No further restrictions. Processing 5 ACL rules.

1 2 3 4 5
5 ACL rules evaluated.

What type of result shall I print?

[w] Whitelist
[b] Blacklist
[s] Shortest of these two
[a] All: Both Whitelist and Blacklist

Which one ?: s

Whitelist for ACL 102 will be relative to the following context
-----

Restrictions as given:

ignore tcp any any established
tcp any any

Restrictions amount to:

tcp *.*.*.* *.*.*.* not established

Whitelist for ACL 102
-----

tcp *.*.*.* 217.31.32.81 eq http not established
tcp 164.47.133-135.* 217.31.32.81 [0-79,81-65535] not established

```

Figure 27

Example application of the 'blackwhite' utility

HTML reformatting after changes to 'HTML_output.conf'

The exact format of the HTML presentation is controlled by the configuration file 'HTML_output.conf' in the 'Output' subdirectory. Whenever the user changes these configuration parameters, a fresh formatting is required. The invocation of this utility has no parameters:

```
perl Utils/make_html.pl
```

With the available configuration options the user may temporarily switch between JavaScript-enabled and pure HTML output format to adjust the encoding to the capabilities of the available HTML browser. Apart from that, the configuration controls mostly the color assignments and the relative size of the various fonts. Depending on the user's equipment, readability may be improved by changing the color settings and the distribution of screen space among the HTML subframes. Further details regarding the configuration options may be found in Chapter 3.5 on page 36.

Creating a comparative display of two evaluation results

The 'make_html' utility may also be used to compare the results of two evaluation runs and to generate a hypertext report with only those results being highlighted that differ between the two evaluations. A typical application of this feature is a reevaluation after a router configuration change, for instance as follows:

1. The user first checks the router configuration — say, the file 'routerconfiguration.cfg' — with an ordinary evaluation run:

```
perl ./crocodile routerconfiguration.cfg
```

2. The user checks the evaluation results and corrects the router configuration 'routerconfiguration.cfg' where appropriate. The new version is then stored in the file 'routerconfiguration-version2.cfg'.
3. In order to analyze the effects of these corrections on the evaluation result, the results of the first evaluation run (stored in 'results.xml') are saved, for example, in the directory '/tmp' under the file name 'firstrun.xml':

```
cp ./XML_data/results.xml /tmp/firstrun.xml
```

4. Next, the user repeats the evaluation run with the corrected configuration 'routerconfiguration-version2.cfg':

```
perl ./crocodile routerconfiguration-version2.cfg
```

5. In the resulting HTML display, the user will see how CROCODILE judges the

new configuration as a whole. However, in this standard display it is hard to recognize in what way the judgement has *changed* due to the corrections. To highlight the effects of the changes only, the user may create a display of the results relative to the earlier, saved evaluation results using the 'make_html' utility by invoking that utility with the name of the reference file:

```
perl Utils/make_html.pl /tmp/firstrun.xml
```

Using the browser, the result of this »differential evaluation« may now be viewed as usual as the 'HTML_Output/index.html' file. The display contains all evaluation results, but only those results that are new or changed from the first to the second evaluation are highlighted. Unchanged results are rendered in neutral coloring — usually in black. A typical example of an evaluation run in full display and in comparative display is shown in Figure 28. A click on 'Source' or 'Reference' in the 'Evaluation Target' window (Figure 28 lower left) provides a comparative side-by-side display of the two underlying configurations if desired. Figure 29 shows this display for the example depicted in Figure 28.

If 'make_html' is invoked once more without parameters, the utility will recreate the usual »absolute« display again, without having to repeat the evaluation run.

7.3 Providing URLs to the IOS Command Reference

The checker tool attempts to assign relevant links to the Cisco IOS Command Reference to each IOS command found in the configuration description. For this purpose, the subdirectory 'Configuration' contains the files 'linkdata.conf' and 'designators.conf' assigning command names to corresponding URLs. The current CROCODILE distribution contains a file 'linkdata.conf' based of IOS release 12.3.

CROCODILE's data set contains only URLs, that is, references to Cisco's command descriptions. For reasons of copyright — but also of disk space — we refrained from including copies of the command references pages. However, this means that the links generated in an evaluation run are only accessible if the results are displayed on a machine with Internet connection. Without Internet access, the referenced IOS pages cannot be loaded into the browser.

Regenerating 'linkdata.conf' requires the following steps:

1. Determine the URL of the »master index« of the desired IOS version. (For IOS 12.3 this is currently the URL <http://www.cisco.com/univercd/cc/td/doc/product/software/ios123/123mindx/crgindx.htm>)
2. Download the master index file from the URL determined in Step 1 to a



Figure 28

Standard display of the evaluation result (top) and differential display relative to an earlier reference result (bottom)

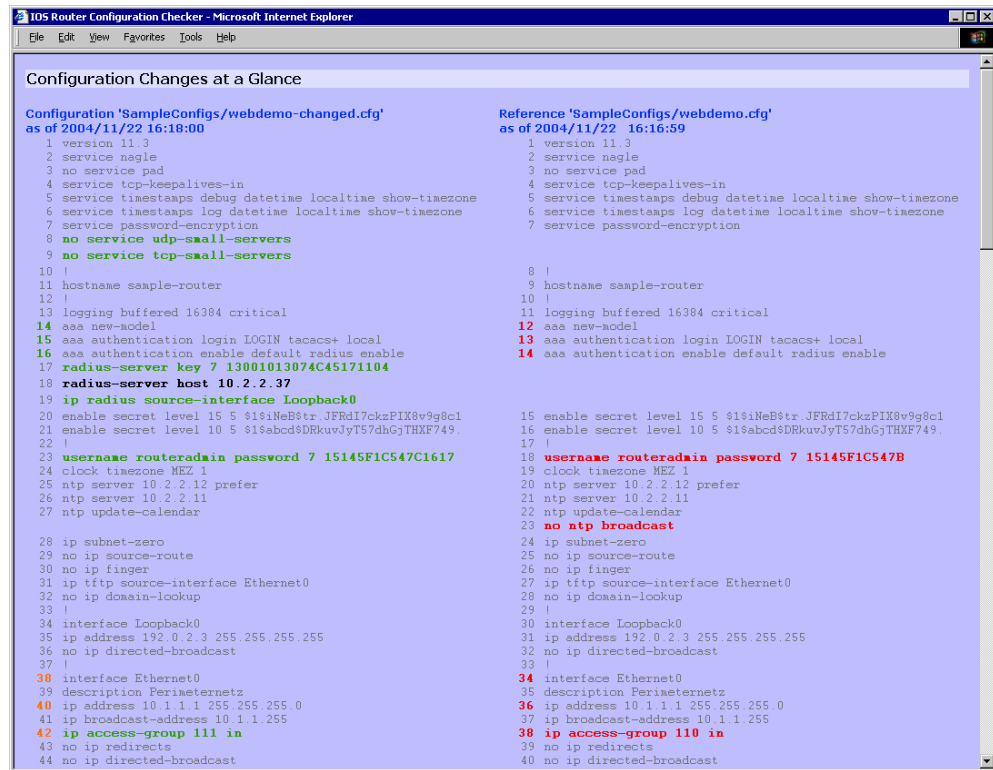


Figure 29

Configuration and evaluation differences side-by-side

temporary local file using the browser.

3. Extract the relevant URLs for each entry from the alphabetical command index and add a corresponding record to the file 'linkdata.conf' in the subdirectory 'Configuration'.

Step 3 is typically done automatically with a suitable filter script.²⁰ The link catalogue 'linkdata.conf' is an ordinary, editable text file. Each line of the file may contain either whitespace, a comment, or an ordered list of text fields as follows:

```
<ios command (prefix)>, {<context>| 'null'}, <Cisco manual name>,  
<URL>, [<Cisco techinfo name>, ... ]
```

²⁰ Earlier version of CROCODILE included the utility 'linkfilter.pl'. Unfortunately, the format of Cisco's IOS Command Reference changed frequently. As we were unable to keep the utility up-to-date, we finally removed 'linkfilter.pl' from the CROCODILE distribution.

The fields must be separated by a comma, and they must appear on the same line without intervening »linefeeds« or »carriage returns«. If there is no information about the context of a command, the second field will contain the string "null". The fifth field is optional: It may reference designators of related Cisco technical documentation. The user may specify an arbitrary number of designators, separated by a comma. See our default 'linkdata.conf' file in the 'Configuration' subdirectory for further illustration.

The list of permissible designators is configured in file 'designators.conf' in the 'Configuration' subdirectory. The user may add new designators by inserting text lines of format

```
<designator> ^ <Short Title> ^ <Full Title> ^ <URL>
```

The »Short Title« will appear as a hyperlink in the evaluation report. Similar to 'linkdata.conf', the list in 'designators.conf' is typically generated semi-automatically with a generator utility. A generator is currently not included in the standard CROCODILE package though.

NOTE: The selection of 'linkdata.conf' and 'designators.conf' is affected by CROCODILE's command line option »crocodile -c <configuration directory> ...«, or the corresponding options »batchcroco -s <ruleset directory> -r <ruleset> ...«, respectively. CROCODILE will first try to read the linkdata and designator configuration files in the configuration subdirectory as defined by -c, -s, and -r options. If no 'linkdata.conf' and 'designators.conf' files are provided in the specified configuration subdirectory, CROCODILE will next try to fetch these files from the default configuration subdirectory '<CROCOHOME>/Configuration'. In most cases there is no need to duplicate the link and designator files across multiple configuration profiles, and the user can simply rely on the default configuration directory.

The user is free to add or delete lines at will to 'linkdata.conf' and 'designators.conf'. There is no need for completeness: If an IOS command is not listed in the catalogue, CROCODILE will simply be unable to add a corresponding link in the evaluation report. Apart from that, the quality of the link catalogues has no essential effect on the analysis results of the tool, it only increases its ease of use.

NOTE: Depending on Cisco's update policy for technical documentation, the links in 'linkdata.conf' and 'designators.conf' may remain more or less stable. Changes of Cisco's web appearance may invalidate our link collection.

8 Creating Custom Checker Modules

CROCODILE is an extensible tool. The user may add custom checker modules — so-called »plug-ins« — to increase the scope and depth of the evaluation. Using the 'Logging' checker module as an example, we will now sketch the recommended procedure to develop custom checker modules.

The purpose of this chapter is to give an impression of the required development effort. CROCODILE users who do not intend to develop custom modules may skip Chapter 8.

8.1 Information gathering

First of all, the developer has to decide on an evaluation topic; in our example, we aim at the configuration settings related to »Logging«. Some configuration commands pertaining to the topic are usually known; these may serve as entry points to determine all relevant commands using the Cisco online documentation. For example, under

www.cisco.com/univercd/cc/td/doc/product/software/ios122/122mindx/index.htm

the »Cisco IOS Release 12.2 Master Index« may be found, and searching there for »logging command« will return a link to the »Cisco IOS Configuration Fundamentals Command Reference«. A look at the table of contents easily shows that mostly the pages FR-482 to FR-505 contain relevant information about logging configuration options.

A close study of the Command Reference will immediately identify the following, potentially important configuration commands:

```
logging, [default] logging buffered, logging console,  
logging facility, logging history [size], logging linecard,  
logging monitor, logging on, logging rate-limit,  
logging source-interface, logging synchronous, logging trap
```

Apart from that, further important commands referring to logging are mentioned:

```
snmp-server enable traps, snmp-server host, access-list (extended)
```

A subsequent analysis must determine which of these commands are actually relevant for an evaluation.

8.2 Analysis

In order to design evaluation criteria, the user must start with an analysis of the individual command descriptions. Even IOS experts would not normally know all commands and their manifold configuration options off the top of their head. The analysis will therefore first make use of the Cisco Reference Manual, and then decide unclear issues by configuration tests on a Cisco router.

Determining the relevant configuration attributes

From the descriptions in the manual it becomes clear that there are five different methods of logging. Each of these methods has a fixed set of configuration attributes. A systematic listing and classification arrives at the following result regarding the relevant attributes per method:

- console logging:
 - activated/deactivated
 - severity level (»level«)
- monitor logging:
 - activated/deactivated
 - severity level
- buffer logging:
 - activated/deactivated
 - severity level
 - buffer size
- SYSLOG logging:
 - activated/deactivated
 - severity level
 - reporting label (»facility«)

- server address(es)
- SNMP logging
 - activated/deactivated
 - severity level
 - history buffer sizes

For each attribute it must be checked whether any defaults are already defined by IOS. The attributes may then be initialized with the corresponding initial values at the start of the analysis.

Choosing an evaluation strategy

For each configuration command the developer needs to determine from the manual which of the above attributes are influenced by the command. Once this analysis step is completed, the checker module's logic is basically clear:

- For each logging attribute, a variable is created in the checker module that mirrors the status of the corresponding attribute; the initial value is set to the default value or »undefined«, respectively.
- For each configuration command, the developer defines a corresponding pattern, with a pattern handler assigned to it. The handler evaluates the command parameters and updates the values of the concerned attribute variables accordingly.
- Once all configuration commands are processed, the checker module determines the resulting logging status from the final attribute values and generates corresponding evaluation report entries.

Performing these steps is simple in principle, but several difficulties arise in practice. This is mostly due to the frequent ambiguity of the Cisco documentation. It is not always clear which attributes exactly are affected by some configuration option: Does omitting an optional parameter result in resetting the concerned attribute to its default value? Does the default differ among various router models? Are all syntactically correct parameter combinations of a command actually permissible in practice? — These and similar uncertainties complicate the analysis. A final decision frequently requires some testing in an IOS router session.

8.3 Implementing the evaluation logic

The investigations so far have more or less made clear which evaluation steps are necessary and how to proceed for those. Now this evaluation strategy must be implemented in Perl code.

Structure of the checker module

The starting point of every new implementation is a code skeleton that is easily created by »copy and paste« from an existing module. A suitable template is shown in Figure 30.

The code fragment shown is in fact a working checker module; it is executable under CROCODILE if the package name is entered in the line one. All other module properties required for interoperation with the tool framework are inherited from the base class 'Plugin::Module'. However, no patterns have been registered yet: The generic handler routine will thus never be triggered; only the (empty) postprocessing routine 'summary()' will be invoked by the CROCODILE framework at the end of the evaluation run.

The developer must now fill the code template with evaluation logic, step by step according to the strategy sketched in Chapter 8.2.

Representing the router configuration state

For each configuration attribute determined in Chapter 8.2, one local state variable is introduced and initialized with the corresponding default value, if any. For instance:

```
## ----- Local state variables for bookkeeping -----

# Console/terminal logging

my $console_log = 1;           # Logging to console?
my $console_level = 7;
my $monitor_log = 1;           # Logging to monitor lines?
my $monitor_level = 7;

# SNMP logging

my $history_level = 4;         # Minimum level sent to SNMP as trap
my $history_size = 1;         # History buffer size
my $traps_enabled = 0;        # Are SNMP traps sent at all?
...
```

```

package Plugin::<module name>;
use strict;

use Plugin::Module;
use vars qw(@ISA);
@ISA = ("Plugin::Module");

use Auxiliary::Logging qw(:user);
my $macroref = [];

my $patternref = { };

## ----- Local state variables for bookkeeping -----

## ----- Public functions -----

sub new {
    my ($proto,@data) = @_;
    my $class = ref($proto) || $proto;

    # Idempotency: At most one instance is created of this class
    my $self=$class->SUPER::does_exist();
    if (!defined $self) {
        $self = $class->SUPER::new(@data);
        $self->{MACROS} = $macroref;
        $self->{PATTERNS} = $patternref;
        $self->{POSTPROCEDURE} = \&summary;
    }

    return $self;
}

## ----- Module-specific handlers -----

sub generic_handler {
    my ($self, $match) = @_;

    $self->_dump_hashref($match, "_generic_handler");
    $self->add_logical_context($match->{LINE_NO}, "<some context>");
    $self->attach_ciscolinks($match->{LINE_NO}, $match->{MATCH});
}

## ----- Postprocessing specifications -----

sub summary {
    my ($self) = @_;
}

```

Figure 30

Template to create a new checker module

Choosing and testing the patterns

Next, the developer must register a pattern for each relevant IOS command. The Cisco Reference Manual is used as an orientation here. However, the formats found there cannot normally be transferred literally. They usually require some adaptations, for instance for the following reasons:

- Frequently, a single pattern is used to cover both the positive and the negated form of a pattern, for example:

```
logging rate-limit {number|all|console} [except severity]
no logging rate-limit.
```

Combining both formats to a single pattern requires enclosing the command parameters with an additional []-bracketing in order to mark them as optional:

```
[no] logging rate-limit [{number|all|console} [except severity]]
```

- The syntax given in the manual is often misleading, if not wrong! For instance, the correct formulation of the syntax for 'logging rate-limit' is as follows (at least for the IOS router we used to check this):

```
logging rate-limit [console [all]|all] number [except severity]
```

Compare this to the specification in the Cisco Manual on page FR-499.

- Sometimes a pattern is intentionally spelled out more generally than actually required by the correct command syntax. This may simplify the pattern, and is also helpful to uncover typing errors or range errors in the configuration text. For example, using the pattern

```
logging history size NUM
```

CROCODILE will also recognize erroneous configuration commands with invalid size specifications, while

```
logging history size 1-500
```

is more exact, but will not match if the allowed maximum size of 500 is exceeded. Generalizations often pay off to invoke the pattern handler even in the case of such mistakes and to reliably catch the error.

The developer may enter suitable patterns in the code fragment and check their function. To simplify pattern specification, macros (see Chapter 3.2, page 17) may be introduced. For a first test, each pattern is assigned the generic handler 'generic_handler()':

```
my $macroref = [
  INTERFACE => "IF_TYPE IF_NUM",
  ...
];

my $patternref = {
  "[no] logging source-interface [ INTERFACE ]" => {
    HANDLER => [ \&generic_handler ],
```

```

    },
    "logging history size NUM" => {
        HANDLER => [ \&generic_handler ],
    },
    ...
};

```

Once all patterns have been registered correctly, the developer may plug the new checker module into the CROCODILE main program, simply by storing the module's source code in the 'Plugin' subdirectory, and by adding its name to 'plugin.conf' in the 'Configuration' subdirectory. Initial testing is possible now, using sample configurations:

- Are all configuration commands (in all possible variants) actually recognized? Even in negated form?
- Are there any ambiguous patterns, resulting in conflicts that must be solved by pattern modifications (see Chapter 3.2, Section »Conflicts: Greedy or minimal pattern interpretation?« on page 21)?
- Is a meaningful manual reference attached to each recognized command by 'attach_ciscolinks()? If the assigned references are unsatisfactory, the search term may be adjusted, or the link databases 'linkdata.conf' and 'designators.conf' in the 'Configuration' subdirectory must be extended or refreshed (see Chapter 7.3).

Only after all patterns work correctly should the programming of the evaluation logic begin.

Programming the handler logic

Pattern by pattern, the developer now replaces the generic handler 'generic_handler()' by a pattern-specific handler routine. The generic handler code serves as a template here.

Inside the handler the '\$match' parameter provides access to the configuration command whose occurrence had triggered the handler. The evaluation logic usually starts with assigning an analysis context to the »match« by 'add_logical_context()' or 'set_clause_context()'. (The preceding invocation of 'dump_hashref()' in the template of Figure 30 lists the '\$match' data set to the console — see Figure 5 on page 29 — and only provides some orientation during development; in the finished checker module, this invocation will be commented out or deleted.) Apart from that, the handler attempts to assign a URL to the command description using 'attach_ciscolinks()'

Next is usually a decision whether the configuration command occurs in negated form or not. In both cases a correctness check of the command parameters will normally follow. Any irregularities are recorded in the result database. The checker module inherits various methods for recording database entries from its base class 'Plugin::Module' (see Appendix A).

Once all checks have been performed and the IOS configuration command has proven valid, the handler will update the state variables of the checker module according to the effects of the configuration command. A typical handler of the 'Logging' checker module is shown in Figure 31. Depending on the available parameters of an IOS configuration clause, the required correctness and plausibility checks may turn out considerably more complex than in this example.

Creating the postprocessing logic

Only after all lines of the router configuration have been analyzed, it is clear what router state will result from the overall configuration. This is why each pattern

```
sub logging_source {
  # [no] logging source-interface [INTERFACE]
  my ($self, $match) = @_;
  my $line = $match->{LINE_NO};

  #-   $self->_dump_hashref($match, "logging_source");

  $self->add_logical_context($line, "Logging");
  $self->add_logical_context($line, "SYSLOG");
  $self->attach_ciscolinks($line, "logging source-interface");

  if (defined $match->{0}) {
    $source_interface = undef;
    if (defined $match->{3}) {
      $self->add_annotation($line, "no params expected", 'WARN');
    }
  }
  elsif (defined $match->{3}) {
    $source_interface = $match->{3};
    $self->change_criticality($line, 'OKAY');
  }
  else {
    $source_interface = undef;
    $self->add_annotation($line, "parameter expected", 'ALERT');
  }
}
```

Figure 31

Example of a handler for the pattern '[no] logging source-interface [INTERFACE]'

handler can contribute only preliminary findings to the result. For a final judgement, a special routine is invoked. Its name is stored in the checker module under '\$self->{POSTPROCEDURE}'. In our example, 'summary()' fills this function.

The postprocessing analyzes the final values of the state variables and draws conclusions from this regarding the configuration state of the router. The resulting positive or negative results are added to the result database. The developer can write into the database with the methods described in Appendix A.

Figure 32 shows a typical excerpt from a postprocessing routine. For further illustration, the reader is referred to the source code of the 'Logging' checker module in 'Plugin/Logging.pm'.

The postprocessing logic completes the checker module. CROCODILE now covers a new evaluation topic. The developer should test the new module extensively with test cases and examples from the real world before releasing it.

```
sub summary {
    my $self = shift;

    if (! defined $linecard_log ) {
        $self->add_context_annotation("Logging",
            "Unclear whether linecard logging is enabled",
            'WARN');
    }
    elsif ($linecard_log == 0) {
        $self->add_context_annotation("Logging",
            "Linecard logging is disabled",
            'INFO');
    }

    ...

    # Create Bookmark label 'Logging' and assign context refs

    $self->add_bookmark("Logging","Logging (General)",
        "Logging", "toContext");
    $self->add_bookmark("Logging", "Log Buffer",
        "Log Buffer", "toContext");
    $self->add_bookmark("Logging", "SYSLOG",
        "SYSLOG", "toContext");
    $self->add_bookmark("Logging", "SNMP Logging",
        "SNMP Logging", "toContext");
}
```

Figure 32

Excerpt from the 'Logging' postprocessing function

Optional: Adding background information to evaluation results

Plugin modules use inherited methods such as 'add_annotation()' or 'add_context_annotation()' to write their evaluation results into CROCODILE's result database. Plugins can write arbitrary text without any constraints on content and length the message; CROCODILE does not enforce any formatting restrictions on the database report messages. Therefore, the result database may contain evaluation results of varying detail and conciseness, depending on the reporting style of the plugin that generated the message.

To achieve a more consistent level of reporting, it is often desirable to base all annotations that are added by the various plugin modules on a common catalogue of evaluation messages: For every possible »evaluation issue« there is predefined, standardized set of evaluation messages that might be generated; each CROCODILE annotation can be looked-up in the catalogue where the user will find further information on the reported issue.

The CROCODILE framework offers message tags to support such a »catalogue of annotations«. A message tag is a unique combination of 2 to 9 capital letters followed by one or more index numbers, separated by dots such as

```
ACL.1.4
ROUTING.4
SNMP.2.13.1
```

The general format of a message tag is

```
/^[A-Z]{2,9}(\.\d+)+\s+$/
```

in Perl regular expression syntax. Whenever such a message tag is put in front of an evaluation message (e.g., a line or context annotation, an »occurrence« or an »absence« message in a compound pattern), CROCODILE will generate a hypertext link for each message tag in the HTML evaluation report referring to file 'messageinfo.<file extension>#<message tag>' in the result directory (i.e., '<CROCOHOME>/HTML_results' in interactive mode). The file extension of this »messageinfo« file is determined by the content of the current »ruleset« subdirectory (in interactive mode '<CROCOHOME>/Configuration' by default): if the ruleset contains a 'messageinfo.*' file, the file extension of that file is used. Furthermore, all files named 'messageinfo.*' and 'messageinfo-aux*' found in the current »ruleset« subdirectory are copied to the result directory.

For example, if the »ruleset« contains two files, 'messageinfo.html' and 'messageinfo-aux.css', these files are copied to 'HTML_results', and a message tag such as 'NTP.3.1' appears in the evaluation report as a hyperlink to 'messageinfo.html#NTP.3.1'.

In this example, we assume that 'messageinfo.html' contains a section explaining the meaning of message 'NTP.3.1', and that the beginning of this section is marked with the appropriate HTML »element ID« tag. File 'messageinfo-aux.css' is assumed to contain a so-called cascading stylesheet describing the presentation of 'messageinfo.html'. Thus, whenever the user clicks a message tag in the HTML evaluation report, the browser will open the ruleset's »messageinfo« file and position the current view at the marked section where additional information about the tagged evaluation message can be found.

Note that CROCODILE's »messageinfo« mechanism is not restricted to HTML files. The ruleset designer may as well provide a message catalogue in PDF format ('messageinfo.pdf'), since many browsers support a '*.pdf#<tag>' notation to navigate directly to the corresponding section of the PDF document. Other file formats (e.g., XML, MS Word, ASCII) are also feasible, even if navigation tags are not supported, as long as the user's browser is able to open the hyperlinked file on request.

Note also that the »messageinfo« mechanism tolerates an arbitrary number of auxiliary files (typically required to describe the formatting and presentation style of the »messageinfo« file). For example, information about evaluation messages may be provided as a set of four files:

- a 'messageinfo.xml' file containing the raw data,
- a 'messageinfo-aux.xml' file describing the attribute selection and HTML formatting of raw data records,
- a 'messageinfo-aux1.css' file describing the presentation's style attributes for screen display, and finally
- a 'messageinfo-aux2.css' file describing the presentation's style attributes for printing.

CROCODILE offers sufficient flexibility for the implementation of quite sophisticated message catalogues.

Logging the course of the evaluation

The checker module records its findings using the methods of the database interface (see Appendix A). Apart from that, some feedback messages about the course of the computation are sometimes required as well. CROCODILE captures such messages in a log file, and restricts console output depending on the verbosity level (option '-v') in effect.

In order to put console output under the control of the »verbosity« settings and to automatically record it in the log file (if any), the developer must use special output operations in the place of Perl's standard 'print'. These operations are provided by the 'user' interface of the auxiliary module 'Auxiliary/Logging.pm' that checker modules must import to gain access to these methods:

```
package Plugin::<module name>;
use strict;
use Auxiliary::Logging qw(:user);
...
```

To record the progress of the computation and to print warning or error messages, the following methods are available:

- console <list>
<list> is written to the console, but not recorded in the log file (equivalent to 'print STDERR <list>, "\n"').
- trace <list>
<list> is written — independent of the »verbosity« level — to the console as well as to the log file.
- info <list>
<list> is written to the log file and, for »verbosity« levels 1 or 2, additionally to the console.
- verbose <list>
<list> is written to the log file and, for »verbosity« level 2, additionally to the console
- warning <list>
<list> is written as a warning (i.e., with '***' preceding each output line) to the log file and to the console.
- error <list>
<list> is written as an error (i.e., with '*** ERROR: ' preceding each output line) to the log file and to the console.
- abort <list>
<list> is written as a warning (i.e., with preceding '*** ' in each output line) to the log file and to the console. After that, the program execution²¹ is aborted with an abort message and a corresponding exit status.

These output methods are implemented such as to be usable exactly like a normal 'print' regarding the syntax (however, without an output file descriptor!). This

implies, for example, that the argument list *<List>* does not need to be enclosed in parentheses, and that arbitrarily many output elements — separated by a comma — may be supplied.

Unlike 'print', however, the above methods automatically append a line feed "\n" at the end of the output. In exceptional cases where this is not desired, the line feed may be suppressed by starting the first list element with a '^', like:

```
warning '^This is part 1 of a multi-part output line; ';
warning 'this is part 2.';
warning 'And here is part 3.';
```

These three commands generate the following output:

```
*** This is part 1 of a multi-part output line; this is part 2.
*** And here is part 3.
```

The messages output with 'warning' or 'error' are automatically counted. The resulting counts are included in the termination messages of the program run. The counts may also be read by the program code at run-time by using 'get_counts':

```
my ($warn_count, $error_count) = get_counts();
```

In order to indicate the progress of the computation, »overwritable« messages may be output using 'Logging.pm': The message text appears on the console only and will be overwritten by the next output command. Two such operations are available:

- `print_eraseable <list>`
<list> is written to the console; the cursor is reset to the start of the line without a line feed (*<list>* must not contain "\n" line feeds!).
- `print_progresswheel`
 A »spinning wheel« in the form of a »spoke« turning in place is output to the console; each invocation will turn the spoke by one position: '|', '/', '-', '\', ...

Overwritable text — whether message or spinning wheel — is overwritten by subsequent messages, by normal as well as overwriting messages. However, in order to make the overwriting and counting of messages work as expected, the

²¹ In batch processing mode, an 'abort' in a checker module or its auxiliary modules will abort only the individual job; the enclosing batch processing front end »catches« the exception and continues processing with the next scheduled job.

methods of 'Logging.pm' must be used exclusively. Commands of the type 'print [STDOUT | STDERR]' should not appear in the finished program code. During the debugging phase of programming, they may of course be safely used.

8.4 Polishing

Proceeding as described in Sections 8.1 to 8.3 will not normally result in a perfect checker module yet. During practical use, modifications and additions will frequently result, for instance:

- The assignment of severity levels (ALERT, WARN, CHECK etc.) often requires some fine tuning in order to achieve a consistent grading of findings across all checker modules.
- Opportunities for additional checks will often arise, for example, by simplifying individual patterns, thus covering additional error cases.
- The set of configuration commands originally deemed relevant sometimes proves incomplete. For example, closer inspection revealed that the following commands are relevant for »logging«, too:

```
mpls traffic-eng logging tunnel, mpls traffic-eng logging lsp
```

 Additional state variables, patterns, and handlers must be provided for these.
- The checker module is also able to analyze whether any configuration clauses cancel each other or whether clauses appear in illegal sequence. This requires a more intricate evaluation logic beyond what can be realized using the straightforward state variables alone.
- The evaluation result may be enriched by (cross) references. Additional logical contexts, bookmarks, dump areas or even fix suggestions may be generated for more information. References to external documentation — possibly in the form of a URL to a white paper on the Internet — are conceivable here.

The developer must decide on the best ratio of benefit and effort. The decision depends not to the least on the degree of expertise the developer assumes the user to have: The more knowledgeable the user, the terser the result presentation.

A look at the original code of the 'Logging' checker module shows how many extensions have been added to round off the checker module.

8.5 Remarks

The 'Logging' checker module is among the simpler, even if lengthier modules of the CROCODILE framework. Its evaluation logic is of moderate complexity. In this respect, this chapter describes a simple example, besides being shortened to the essential aspects. Actually, the design, implementation, and test of the 'Logging' checker module required a few person days of work.

Depending on the evaluation topic, developing a checker module may be considerably more expensive. The factors affecting the development effort include:

- How many configuration commands does the evaluation topic comprise? How extensively may these commands be parameterized?
- How independent are the relevant configuration commands from each other? How many interdependencies must be observed (e.g., a prescribed sequence of configuration steps)?
- How clear, unambiguous, and complete is the documentation provided by Cisco? (Respectively: How good is the developer's expertise in this evaluation topic?)

In particular the last item often turns out as the crucial »cost driver« in the creation of a new checker module, as the Cisco documentation too often leaves important questions unanswered. At times, the documentation is actually misleading! Extensive router tests are required then in order to achieve clarity regarding the actual effects of certain configuration commands.

If the developer is well-versed in the evaluation topic, the procedure presented in this chapter should result — after a little routine — in a fast and systematic development of customized checker modules.

9 Status and Outlook

9.1 Status of the tool

The CROCODILE framework has proven its usefulness so far, even though it has not been completed in every respect. In most practical configurations analyzed using CROCODILE we have found at least irregularities — often even severe errors! If our observations are representative, configuration defects are the norm rather than the exception, and security breaches often remain undetected for months. The fact that we discovered with modest means defects overlooked even by trained staff confirms our approach.

The generic checker module 'CompoundPatterns' contributes considerably to CROCODILE's overall coverage of router security. Security requirements of the most varied kind have shown to be easily expressible in the 'CompoundPatterns' formalism. The approach of introducing a simple meta-language for the declaration of evaluation criteria has proven rather helpful here.

'CompoundPatterns' emphasizes the simplicity of the rule declarations, which limits the analytical power and reporting precision of this checker module. A more complex specification language for evaluation criteria would be more versatile, but also requires deeper knowledge and more effort on the part of the user. Only practice can reveal the most balanced compromise between power and ease of use of such a generic module. The 'IngressEgress' module demonstrates how evaluation requirements that are both flexible and powerful may be combined with ease of use — if the evaluation scope is suitably narrowed.

The output format to display the evaluation results is rather mature in our view. Making use of HTML hypertext enables a flexible adaptation of the format to the needs of the user; requiring no more than a standard browser ensures very good portability to the most diverse platforms. In batch processing mode, CROCODILE optionally delivers its result in CSV format as well, enabling simple postprocessing of the data with external tools.

9.2 Next steps

The available checker modules cover only some — though important — aspects of router security. Existing gaps in the evaluation logic that cannot be treated

completely with the simple means of 'Compound Patterns' may be closed by additional checker modules. Further, it is desirable to generate very specific diagnostic messages for individual defects, which is not possible using 'CompoundPatterns'.

Several evaluation topics will probably require configurable checker modules that must be equipped with a module configuration in a suitable format. Future examination will have to show how the user can specify evaluation criteria for certain topics in the most sensible way.

The CROCODILE framework offers an excellent basis for an extension of the evaluation scope. However, the required implementation effort can only be justified by a corresponding user demand: Covering evaluation topics only »just in case« is a viable option only for a few core topics.

On the other hand, every new checker module benefits all CROCODILE users, as it can be made available to all the user community. For the sake of all users, we therefore hope for a sufficiently large customer base ensuring a cost-effective future development of our framework.

9.3 Extension to other network components

CROCODILE is sufficiently flexible to be applied to other configuration formats. At the time of this writing, a new plugin module for the dataflow analysis of Cisco PIX/ASA firewall configurations is under development. Cisco firewalls use a somewhat different, but rather similar command logic compared to IOS. While a complete redesign of the patterns and pattern handlers is required, the structure of the tool environment, including parser, utilities, and HTML output, remains unchanged and will be reused.

Network devices from other vendors than Cisco can be analyzed with CROCODILE, provided their configuration is available as a textual description with sensible syntax rules. For example, we are about to implement new plugins and rulesets for the Redback SmartEdge family of routing devices. So far, the CROCODILE framework nicely supports the analysis of SmartEdge OS configurations.

A Interface to the Result Database

The following description of the interface between a checker module and the result database addresses only those users who intend to modify existing checker modules or plan to develop new ones.

All checker modules store their results in the central result database. The database is realized by the Perl module 'Auxiliary/ResultData.pm'. It provides methods to store the various result data types: Annotations, result categories, suggestions etc. — details may be found in Chapter 3.4 on page 29.

Database interface of the checker modules

All access methods of the result database are write operations. Querying the result database is not supported. This simplifies the interfaces and the interaction between the modules and the database.

Checker modules use the database interface usually not directly, but through wrapper methods with a simplified parameter list, provided by their common base class 'Plugin::Module'. These methods are²²:

- `set_clause_context($linenumber, $context_name)`
The method assigns a clause context to a line recognized by the parser. Each line may belong to only one clause context, as implied by the IOS configuration modes. In choosing a name for the context, it is recommended to adhere to the convention of using the line number of the configuration clause opening the context (e.g., the line starting with the keyword 'interface' for all configuration clauses of the respective interface). A context does not need to be created explicitly, but is created automatically as soon as it is assigned to a line.
- `change_criticality($linenumber, $severity [, $force_flag])`
This changes the severity of a line recognized by the parser. However, this

²² Please note that the concept introduced in this manual as the »severity« of a finding used to be called »criticality« in an earlier version of CROCODILE. This change of name, however, is not yet fully reflected in the source code of CROCODILE, including the method names of the database interface described in this appendix.

method is normally executed implicitly by invoking 'add_annotation', so that an explicit invocation is rather the exception. Nevertheless, it is important to understand the handling of severities.

Every module indicates its individual severity estimation of a configuration clause. Only the output module will finally determine what the overall severity for this line should be, after observing all individual module estimates. However, each module must decide on one estimate per line. If this method is invoked several times by the same module with differing severity for the same line, the new value will not overwrite the previous one, but the more severe one will be recorded (i.e., ALERT before WARN etc.). However, it may happen that a »blind« overwriting of the previous value is just what is desired, such that the new severity should have effect in any case. In such a case, the 'force' flag may be set (i.e., the last argument is set to an arbitrary value). Note that a severity indicated like this, while having effect after this invocation, may later be replaced again by the more severe of two values if 'change_criticality()' is invoked once more without the 'force' flag. Legal values for 'severity' are 'OKAY', 'INFO', 'CHECK', 'WARN' or 'ALERT' (ordered from the least to the most severe value here).

- `add_annotation($linenumber, $text, $severity)`
This method stores an annotation for a line. Every invocation of the method will append a new annotation to the line in the result database. The indicated severity is assigned to the annotation. Invoking 'add_annotation()' implies 'change_criticality()' such that adding an annotation will also change the severity of the line.
- `add_reference($linenumber, $ref_name, $reference, $ref_type)`
This method assigns a reference to a configuration clause. Every line may have an arbitrary number of references. The arguments beside the line number include the visible name for the reference, the reference (e.g., a URL or a file name) and the reference type. Possible types are 'file', 'URL', 'DumpData', 'toLine' and 'toContext'. The first type is a path name to a local file, the second links to a URL, the third points to a text of type DumpData ('\$reference' must be the name of the DumpData object in the result database). A reference of type 'toLine' refers to an arbitrary line in the router configuration text, and one of type 'toContext' refers to a context (with '\$reference' indicating the context name), which may be a logical context as well as a clause context. With the last three types, it is important that the referenced data object has been created before the reference is created. For example, it is not possible to refer to a line that has not (yet) been recognized by the parser. Contexts and dumps must have been created before in the result database as well.
- `add_logical_context($linenumber, $context_name)`
This method assigns an arbitrary logical context to a line in order to make a

relationship between various configuration clauses explicit. The context name may be arbitrary. A context does not need to be created explicitly, but is created automatically as soon as it is assigned to a line.

- `add_fix_suggestion($linenumber, $suggestion, $comment)`
This method assigns a fix suggestion to a line. Every new invocation of this method appends another suggestion. Such a fix suggestion normally consists of a code fragment and an explanation. The parameter '\$suggestion' indicates the code, '\$comment' the explanation.
- `change_context_criticality($context_name, $severity [, $force_flag])`
This method changes the severity of an existing context. See the explanation of 'change_criticality()' above.
- `add_context_annotation($context_name, $text, $severity)`
This method assigns an annotation to a clause context or logical context. See the explanation of 'add_annotation()' above.
- `add_context_ref($context_name, $ref_name, $reference, $ref_type)`
This method assigns a reference to a clause context or logical context. See the explanation of 'add_reference()' above.
- `rename_context($old_context_name)`
It happens that a context must be renamed. For example, this is the case if a context named »accesslist 102« had been created, but that ACL is redefined in the course of the configuration description. In this case, the old context »accesslist 102« may be renamed by the system. Other lines may be assigned to this context name then. The new name of the old context is automatically determined by the system and returned as the result of this method. This way, annotations etc. may still be added to the »old« context.
- `add_DumpData($name, $position, $text)`
This method stores formatted text. Each block of text is named '\$name' and may be referenced using that name. The text is displayed as a table. This is why a row and column number must be indicated with every text string to be entered into the dump text block, giving the position where in the block the new text should appear. If an output spanning a whole line is desired, an asterisk may be used instead of a column number. Both values are specified in the position string: First, a row number is expected, then — separated by a comma — the column number (or an asterisk). A position string intended to write a text into the second column of the fourth row would thus look as follows: "4,2". If the subsequent fifth line shall not be separated into columns, but written contiguously (e.g., for a heading), "5,*" must be used. The position suffix ",*" is optional and may be omitted, that is, "5,*" is equivalent to "5". The column size of the table is computed automatically

from the lengths of the text strings written into the columns. If a contiguous line (i.e., a line spanning multiple columns) is desired that should not begin in the first, but, for example, in the third column, "6,3*" will enter a text string in the sixth row starting at the third column and extending across all following columns, not affecting their sizes. This may be used, for example, for an indented heading. In order to fill several columns of a column with text, the method must be invoked several times. Each dump may be modified by one module only.

- `add_bookmark($name, $ref_name, $reference, $ref_type [, $severity])`
This method assigns a reference to a bookmark category. Bookmarks are mostly used as »containers« for findings that do not belong to a specific configuration clause, nor to a single clause context or logical context. In the extended HTML display, there is a bookmark frame in the upper left listing all bookmarks under their assigned '\$name'. They are nothing but a collection of references, to be handled like all other references. See the explanations of 'add_reference()' above.
- `attach_ciscolinks($linenum, $commandstring [, $context])`
This method assigns a URL reference to a configuration line, pointing to the Cisco online documentation (see Figure 6, page 33, lower left). The hyperlinks refer to syntax and function descriptions of the command. The '\$commandstring' is usually simply the literal configuration clause. The method will search a specially prepared database for URLs matching that command. Sometimes it is necessary to resolve ambiguity by restricting the search to a specific clause context (typically, the enclosing IOS configuration mode) to prevent the search heuristics from returning too many unrelated URLs. The URLs for the commands are stored in the text files 'linkdata.conf' and 'designators.conf' in the 'Configuration' subdirectory. More details on this may be found in Chapter 7.3.

For examples of how these methods are applied in practice, the reader is referred to the source code in the 'Plugin' subdirectory.

References

CIS	Center for Internet Security (CIS) http://www.cisecurity.org
CiscoTech	Cisco Systems Inc.: Improving Security on Cisco Routers. 1992-2001 http://www.cisco.com/warp/public/707/21.html
IOS12.0a	Cisco Systems Inc.: Cisco IOS 12.0 Network Security. Cisco Press 1999
IOS12.0b	Cisco Systems Inc.: Cisco IOS 12.0 Configuration Fundamentals. Cisco Press 1999
IOSessentials	Cisco Systems Inc.: Essential IOS Features Every ISP Should Consider. Version 2.9, Juni 2001 http://www.cisco.com/public/cons/isp/documents/IOSEssentialsPDF.zip
NSA	U.S. National Security Agency. http://www.nsa.gov
NSAGuide	National Security Agency: Router Security Configuration Guide (Version 1.1b). NSA Report No. C4-040R-02, Dezember 2003 http://www.nsa.gov/snac/routers/cisco_scg-1.1b.pdf
RAT2.2	Center for Internet Security (CIS): CIS Level-1 / Level-2 Bench- marks and Audit Tool for Cisco IOS Routers and PIX firewalls. RELEASE 2.2, September 2004 http://www.cisecurity.org/bench_cisco.html
RFC1918	Network Working Group: Address Allocation for Private Internet. Request for Comments No. 1918, Februar 1996 http://www.faqs.org/rfcs/rfc1918.html
SANS	The SANS (SysAdmin, Audit, Network, Security) Institute http://www.sans.org

List of Abbreviations

AAA	Authentication, Authorization, and Accounting
ACL	Access [Control] List
ASCII	Americal Standard Code for Information Interchange
BGP	Border Gateway Protocol
BNF	Backus-Naur Form
CIS	Center for Internet Security
CSS	Cascading Style Sheet (HTML)
CDP	Cisco Discovery Protocol
CPAN	Comprehensive Perl Archive Network
CROCODILE	Cisco Router Configuration Diligent Evaluator
EIGRP	Extended Interior Gateway Protocol
FTP	File Transfer Protocol
HREF	Hypertext Reference (HTML)
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IOS	[Cisco] Internetworking Operating System
IP	Internet Protocol
ISP	Internet Service Provider
IT	Information Technology
MIB	Management Information Base
MPLS	Multi-Protocol Label Switching
MS	Microsoft
NAT	Network Address Translation
NSA	(US) National Security Agency
NTP	Network Time Protocol
PDF	Portable Document Format
Perl	Practical Extraction and Report Language (resp.: Pathologically Eclectic Rubbish Lister)
PIX	Private Internet Exchange (Cisco Firewall)
PPP	Point-to-Point Protocol
RADIUS	Remote Authentication Dial-in User Service
RAT	Router Auditing Tool
RFC	Request for Comments
SANS	Sysadmin, Audit Network, Security (Institute)
SNMP	Simple Network Management Protocol

TACACS	Terminal Access Controller Access Control System
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TTY	Teletypewriter (Interface)
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VIP	Versatile Interface Processor
VTY	Virtual TTY (Interface)
WWW	World-wide Web
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language

Document Information

Title:	Crocodile® User Manual
Date:	October 2008
Report:	011.04/E
Status:	Version 3.4

Copyright © 2008, Fraunhofer IESE.

CROCODILE ® is a registered trademark of
Fraunhofer IESE.

